
Documentation Perl

(en français)

version du 6 août 2009

Chapitre 1

Avant-propos

Ce document est la traduction (encore incomplète) de la documentation originale distribuée avec perl. Ces traductions ont été réalisées et relues par de nombreux volontaires. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant, entre autres, l'URL <<http://www.enstimac.fr/Perl/>>.

Cette compilation en un seul document PDF a été conçue en utilisant une version francisée du script Perl `pod2latex` pour obtenir du L^AT_EX puis convertie en PDF grâce à `pdflatex`. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message (Paul.Gaborit @ enstimac.fr) mais rien ne vous y oblige.

Si vous avez des remarques concernant ces documents (chaque chapitre est un document spécifique), en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Certaines parties ne sont toujours pas traduites. D'autres n'ont pas encore été relues. La plupart ne sont pas à jour par rapport à leur version anglaise originale (la version de référence de la traduction est indiquée dans chaque chapitre). Certains outils n'ont pas encore été francisés. Toutes les bonnes volontés sont donc les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.

Le 6 août 2009 – Paul Gaborit

Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs des différents documents originaux (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la compilation PDF de ces documents.

Première partie

Présentation

Chapitre 2

perl

Langage pratique d'extraction et de rapport

2.1 SYNOPSIS

```
perl [ -sTtuUWX ] [ -hv ] [ -V[:configvar] ] [ -cw ] [ -d[t][:debugger] ] [ -D[number/list] ] [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -O[octal/hexadecimal] ] [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ] [ -f ] [ -C [number/list] ] [ -P ] [ -S ] [ -x[dir] ] [ -i[extension] ] [ -e 'command' ] [ - ] [ programfile ] [ argument ]...
```

2.2 DESCRIPTION

Perl est un langage optimisé pour extraire des informations de fichiers texte et imprimer des rapports basés sur ces informations. C'est aussi un bon langage pour de nombreuses tâches d'administration système. Il est écrit dans le but d'être pratique (simple à utiliser, efficace, complet) plutôt que beau (petit, élégant, minimaliste).

Perl combine (du point de vue de l'auteur) les meilleures fonctionnalités de C, **sed**, **awk** et **sh**, de manière telle que les personnes familières de ces langages ne devraient avoir aucune difficulté avec celui-ci. (Les historiens pourront aussi noter quelques vestiges de **csh**, de Pascal et même de BASIC-PLUS). La syntaxe se rapproche beaucoup de celle du C. Contrairement à la plupart des utilitaires Unix, Perl ne limite pas arbitrairement la taille des données – si vous avez assez de mémoire, Perl peut copier l'intégralité d'un fichier dans une seule chaîne de caractères. Il n'y a pas de niveau maximum à la récursivité. Et les tables utilisées par les tables de hachage (appelées aussi "tableaux associatifs") croissent dès que nécessaire afin de garantir un bon niveau de performance. Perl utilise des techniques sophistiquées de recherche de motifs pour pouvoir traiter très rapidement de très grandes quantités de données. Bien qu'optimisé pour le traitement des fichiers textes, Perl peut aussi traiter des données binaires et faire que des fichiers DBM soient vus comme des tables de hachage. Les scripts Perl ayant leurs setuid bits positionnés sont plus sûrs que des programmes C grâce à des mécanismes de suivi de flot de données qui permettent d'éviter de nombreux trous de sécurité particulièrement stupides.

Face à un problème pour lequel vous auriez habituellement utilisé **sed**, **awk** ou **sh**, mais qui dépasse leurs capacités ou qui doit fonctionner un peu plus rapidement et pour lequel vous ne voulez pas écrire en C, alors Perl est pour vous. Il existe aussi des convertisseurs pouvant transformer vos scripts **sed** et **awk** en scripts Perl.

Si vous débutez en Perl, vous devriez commencer par la lecture de *perlintro* qui est une introduction générale destinée aux débutants et qui fournit les éléments nécessaires à une bonne navigation dans la documentation complète de Perl.

Pour en simplifier l'accès, le manuel Perl a été scindé en plusieurs sections :

2.2.1 Présentation

perl	Vue d'ensemble (ce document)
perlintro	Brève introduction et vue d'ensemble de Perl
perltoc	Table des matières de la documentation Perl

2.2.2 Tutoriels

perlreftut	Le très court tutoriel de Mark sur les références
perldsc	Livre de recettes des structures de données en Perl
perllol	Manipulation des tableaux de tableaux en Perl
perlrequick	Les expressions rationnelles Perl pour les impatientes
perlretut	Tutoriel des expressions rationnelles en Perl
perlboot	Tutoriel pour l'orienté objet à destination des débutants
perltoot	Tutoriel orienté objet de Tom
perltooc	Le tutoriel de Tom pour les données de classe OO en Perl
perlbot	Collection de trucs et astuces pour Objets (the BOT)
perlstyle	Comment (bien) écrire du Perl
perlcheat	Anti-sèche Perl 5
perltrap	Les pièges de Perl pour l'imprudent
perldebtut	Tutoriel de débogage de Perl
perlfaq	Foire aux questions sur Perl
perlfaq1	Questions d'ordre général sur Perl
perlfaq2	Trouver et apprendre Perl
perlfaq3	Outils de programmation
perlfaq4	Manipulation de données
perlfaq5	Fichiers et formats
perlfaq6	Expressions rationnelles
perlfaq7	Problèmes du langage Perl
perlfaq8	Interaction avec le système
perlfaq9	Réseau

2.2.3 Manuel de référence

perlsyn	Syntaxe de Perl
perldata	Types de données de Perl
perlop	Opérateurs Perl et priorité
perlsub	Les sous-programmes de Perl
perlfunc	Fonctions Perl prédéfinies
perlopentut	Tutoriel de la fonction Perl open()
perlpacktut	Tutoriel des fonctions Perl pack() et unpack()
perlpod	Plain old documentation (« bonne vieille documentation »)
perlpodspec	Plain old documentation, format et spécification
perlrun	Comment utiliser l'interpréteur Perl
perldiag	Les différents messages de Perl
perllexwarn	Les avertissements de Perl et leur contrôle
perldebug	Débogage de Perl
perlvar	Variables prédéfinies en Perl
perlre	Les expressions rationnelles en Perl
perlrebackslash	Les séquences backslash des expressions rationnelles
perlrecharclass	Les classes de caractères des expressions rationnelles
perlrefref	Résumé rapide des expressions rationnelles en Perl
perlref	Références et structures de données imbriquées en Perl
perlform	Formats Perl
perlobj	Objets de Perl
perltie	Comment cacher un objet d'une classe derrière une simple variable
perlDBMfilter	Filtres DBM en Perl
perlipc	Communication inter-processus en Perl (signaux, files d'attente, tubes, sous-processus sûrs, sockets et sémaphores)
perlfork	Information sur fork() en Perl
perlnumber	Sémantique des nombres en Perl

perlthrtut	Tutoriel sur les threads en Perl
perlothrtut	Ancien tutoriel sur les threads en Perl
perlport	Écrire du code Perl portable
perllocale	Gestion des "locale" en Perl (internationalisation et localisation)
perluniintro	Introduction Unicode en Perl
perlunicode	Utilisation Unicode en Perl
perlunifaq	FAQ Unicode en Perl
perlunitut	Tutoriel Unicode en Perl
perlebcdic	Comment utiliser Perl sur des plateformes EBCDIC
perlsec	Sécurité de Perl
perlmod	Modules Perl (paquetages et tables de symboles)
perlmodlib	Pour construire de nouveaux modules et trouver les existants
perlmodstyle	Comment écrire correctement des modules en Perl
perlmodinstall	Installation des modules CPAN
perlnewmod	Préparer un module en vue de sa distribution
perlpragma	Les paquetages utilitaires de la distribution Perl
perlutil	Utilitaires intégrés dans la distribution Perl
perlcompile	Introduction à la compilation Perl
perlfilter	Filtres de source Perl
perlglossary	Glossaire Perl

2.2.4 Implémentation et interface avec le langage C

perlembed	Utiliser Perl dans vos programmes en C ou C++
perldebguts	Les entrailles du débogage de Perl
perlxstut	Guide d'apprentissage des XSUB
perlxs	Manuel de référence du langage XS
perlclib	Fonctions internes se substituant aux fonctions C standard
perlguts	Fonctions internes pour réaliser des extensions
perlcall	Conventions d'appel de Perl depuis le C
perlreapi	Interface des greffons du moteur d'expressions rationnelles
perlreguts	Les entrailles du moteur d'expressions rationnelles
perlapi	Liste des API Perl (générée automatiquement)
perlintern	Fonctions internes Perl (générée automatiquement)
perliol	API C pour utiliser les filtres d'E/S de Perl
perlpio	Interface d'abstraction des E/S internes à Perl
perlhack	Guide des hackers en Perl

2.2.5 Divers

perlbook	Livres concernant Perl
perltodo	Ce qui reste à faire pour Perl
perldoc	Consulter la documentation Perl au format Pod

perlhists	Les archives de l'histoire de Perl
perldelta	Nouveautés de la dernière version
perl595delta	Nouveautés de la version 5.9.5
perl594delta	Nouveautés de la version 5.9.4
perl593delta	Nouveautés de la version 5.9.3
perl592delta	Nouveautés de la version 5.9.2
perl591delta	Nouveautés de la version 5.9.1
perl590delta	Nouveautés de la version 5.9.0
perl588delta	Nouveautés de la version 5.8.8
perl587delta	Nouveautés de la version 5.8.7
perl586delta	Nouveautés de la version 5.8.6
perl585delta	Nouveautés de la version 5.8.5
perl584delta	Nouveautés de la version 5.8.4
perl583delta	Nouveautés de la version 5.8.3
perl582delta	Nouveautés de la version 5.8.2
perl581delta	Nouveautés de la version 5.8.1
perl58delta	Nouveautés de la version 5.8
perl573delta	Nouveautés de la version 5.7.3
perl572delta	Nouveautés de la version 5.7.2
perl571delta	Nouveautés de la version 5.7.1
perl570delta	Nouveautés de la version 5.7.0
perl561delta	Nouveautés de la version 5.6.1
perl56delta	Nouveautés de la version 5.6
perl5005delta	Nouveautés de la version 5.005
perl5004delta	Nouveautés de la version 5.004
perlartistic	Licence artistique de Perl
perlgpl	Licence publique générale GNU

2.2.6 Spécificités pour certaines langues

perlcn	Perl et le chinois simplifié (en EUC-CN)
perljp	Perl et le japonais (en EUC-JP)
perlko	Perl et le coréen (en EUC-KR)
perltw	Perl et le chinois traditionnel (en Big5)

2.2.7 Spécificités pour certaines plateformes

perlaix	Perl sur plateforme AIX
perlamiga	Perl sur plateforme AmigaOS
perlapollo	Perl sur plateforme Apollo DomainOS
perlbeos	Perl sur plateforme BeOS
perlbs2000	Perl sur plateforme POSIX-BC BS2000
perlce	Perl sur plateforme WinCE
perlcygwin	Perl sur plateforme Cygwin
perldgux	Perl sur plateforme DG/UX
perldos	Perl sur plateforme DOS
perlepoc	Perl sur plateforme EPOC
perlfreesbsd	Perl sur plateforme FreeBSD
perlhpx	Perl sur plateforme HP-UX
perlhurd	Perl sur plateforme Hurd
perlirix	Perl sur plateforme Irix
perllinux	Perl sur plateforme Linux
perlmachten	Perl sur plateforme Power MachTen
perlmacos	Perl sur plateforme Mac OS (Classic)
perlmacosx	Perl sur plateforme Mac OS X
perlmint	Perl sur plateforme MiNT
perlmpaix	Perl sur plateforme MPE/iX
perlntware	Perl sur plateforme NetWare
perlos2	Perl sur plateforme OS/2

perlos390	Perl sur plateforme OS/390
perlos400	Perl sur plateforme OS/400
perlplan9	Perl sur plateforme Plan 9
perlqnx	Perl sur plateforme QNX
perlrisco	Perl sur plateforme RISC OS
perlsolaris	Perl sur plateforme Solaris
perlsymbian	Perl sur plateforme Symbian
perltru64	Perl sur plateforme Tru64
perluts	Perl sur plateforme UTS
perlvesa	Perl sur plateforme VM/ESA
perlvm	Perl sur plateforme VMS
perlvos	Perl sur plateforme Stratus VOS
perlwin32	Perl sur plateforme Windows

Sur les systèmes Debian et dérivés, vous devrez installer le paquetage **perl-doc**. Il contient la majeure partie de la documentation Perl standard et le programme *perldoc*.

Une documentation complète est disponible dans les modules Perl, tant pour les modules distribués directement avec Perl que pour les modules tiers empaquetés pas ailleurs ou installés localement.

Vous devriez être en mesure de lire la documentation Perl avec la commande `man(1)` ou via `perldoc(1)`.

Si votre programme a un comportement étrange et que vous ne savez pas où se trouve le problème, pour demander à Perl de vous aider, utilisez l'option `-w`. Très souvent Perl vous indiquera l'endroit exact où se trouve le problème.

Mais il y a plus encore...

Débutée en 1993 (voir *perlhst*), la version 5 de Perl constitue une réécriture presque complète et introduit les fonctionnalités suivantes :

- Modularité et réutilisabilité via d'innombrables modules
Voir *perlmod*, *perlmodlib* et *perlmodinstall*.
- Intégration et extension
Voir *perlembed*, *perlxtst*, *perlx*, *perlcall*, *perlguts* et *xsubpp*.
- Création de vos propres variables magiques (incluant l'accès simultané à plusieurs implémentations de DBM)
Voir *perltie* et *AnyDBM_File*.
- Subroutines surchargées, à chargement automatique et prototypées
Voir *perlsub*.
- Structures de données imbriquées et fonctions anonymes
Voir *perlrefut*, *perlref*, *perldsc* et *perllol*.
- Programmation orientée objet
Voir *perlobj*, *perlboot*, *perltoot*, *perlooc* et *perlbot*.
- Support de processus légers (threads)
Voir *perlthrtut* et *threads*.
- Gestion d'Unicode, de l'internationalisation et de la localisation
Voir *perluniintro*, *perllocale* et *Locale::Maketext*.
- Portée lexicale
Voir *perlsub*.
- Amélioration des expressions rationnelles
Voir *perltre* avec des exemples dans *perlop*.
- Débogueur amélioré et intégration des échanges avec les éditeurs de source
Voir *perldebut*, *perldebug* et *perldebuts*.
- Bibliothèque conforme POSIX 1003.1
Voir *POSIX*.

Ok, terminons-là le battage publicitaire en faveur de Perl.

2.3 DISPONIBILITÉ

Perl est disponible sur la plupart des systèmes et cela inclut quasiment tous les systèmes de type Unix. Voir PLATEFORMES in *perlport* pour une liste.

2.4 ENVIRONNEMENT

Voir *perlrun*.

2.5 AUTEUR

Larry Wall <larry@wall.org>, aidé par de nombreuses autres personnes.

Si vous désirez faire partager votre témoignage sur des succès remportés en utilisant Perl, aidant ainsi ceux qui voudraient recommander Perl pour leurs applications, ou tout simplement si vous voulez exprimer votre gratitude à Larry et l'équipe de développement, alors écrivez s'il vous plaît à <perl-thanks@perl.org>.

2.6 FICHIERS

"@INC" emplacements des bibliothèques Perl

2.7 VOIR AUSSI

a2p traducteur awk vers perl
s2p traducteur sed vers perl

http://www.perl.org/ le site de Perl
http://www.perl.com/ Articles sur Perl (O'Reilly)
http://www.cpan.org/ les archives complètes de Perl
http://www.pm.org/ les Perl Mongers (groupes d'utilisateurs de Perl)

http://www.mongueurs.net/
 Les Mongueurs de Perl (groupes d'utilisateurs francophones)

2.8 DIAGNOSTICS

La directive `use warnings` (et l'option `-w`) génère de magnifiques diagnostics.

Voir *perldiag* pour l'explication de tous ces diagnostics. La directive `use diagnostics` oblige Perl à produire ces messages dans leur forme longue.

En cas d'erreur de compilation le numéro de la ligne fautive est indiqué ainsi que l'emplacement approximatif du mot concerné. (Dans le cas de script fourni par l'option `-e`, chaque utilisation de `-e` est comptée comme une ligne.)

Les scripts en setuid ont des contraintes supplémentaires pouvant produire des messages d'erreur tel que "Insecure dependency". Voir *perlsec*.

Avons-nous mentionné que vous devriez vraiment penser à utiliser l'option `-w` ?

2.9 BUGS

Le modificateur `-w` n'est pas obligatoire.

Le fonctionnement de Perl dépend de la manière dont votre machine implémente certaines opérations telles que le changement de types, la fonction `atof()` ou l'affichage des nombres flottants par la fonction `sprintf()`.

Si votre stdio nécessite un déplacement (`seek`) ou un test de fin de fichier (`eof`) entre les lectures et les écritures sur un même flot de données, alors Perl le requiert aussi. (Ceci ne s'applique pas à `sysread()` et `syswrite()`.)

Il n'y a aucune limite en ce qui concerne la taille des types de données prédéfinis (à part la mémoire disponible) mais il existe toujours un petit nombre de limites arbitraires : un nom de variable ne peut dépasser 251 caractères. Les numéros de lignes affichés par **diagnostics** sont stockés en interne dans des entiers courts et sont donc limités à 65535 (les nombres plus grands sont généralement traités cycliquement).

Vous pouvez envoyer vos rapports de bug (assurez-vous d'inclure toutes les informations sur la configuration obtenue par le programme `myconfig` fourni avec Perl, ou par `perl -V`) à <perlbug@perl.com>. Si vous avez réussi à compiler Perl, le script `perlbug` fourni dans le répertoire `utils/` peut être utilisé pour envoyer un rapport de bug.

L'acronyme Perl signifie réellement Pathologically Eclectic Rubbish Lister, mais ne dites à personne que j'ai dit ça.

2.10 NOTES

La devise de Perl est "Il y a toujours plus d'une façon de le faire". Devinez exactement combien de façons est laissée en guise d'exercice pour le lecteur.

Les trois grandes vertus du programmeur sont la paresse, l'impatience et l'orgueil. Reportez-vous au livre "Programmation en Perl" (le Camel Book) pour savoir pourquoi.

2.11 TRADUCTION

2.11.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

2.11.2 Traducteur

Marc Carmier <carmier@immortels.frmug.org>. Mise à jour vers Perl 5.10.0 par Paul Gaborit (Paul.Gaborit arobase enstimac.fr).

2.11.3 Relecture

Pascal Ethvignot <pascal@encelade.frmug.org>, Gérard Delafond.

Chapitre 3

perlintro

Brève introduction et vue d'ensemble de Perl

3.1 DESCRIPTION

Ce document a pour but de donner une vue d'ensemble du langage de programmation Perl tout en fournissant quelques pointeurs vers de la documentation complémentaire. Il est conçu comme un guide de démarrage destiné à un public de novices. Il apporte juste assez d'informations pour permettre de lire des programmes Perl écrits par d'autres et de comprendre approximativement ce qu'ils font, ou d'écrire des scripts simples.

Ce document ne cherche pas à être complet. Il ne tente même pas d'être parfaitement exact. Dans certains exemples la perfection a été sacrifiée dans le seul but de faire passer l'idée générale. Il vous est *fortement* recommandé de compléter cette introduction par la lecture du manuel de Perl, dont la table des matières peut être trouvée dans *perltoc*.

Tout au long de ce document vous découvrirez des références à différentes parties de la documentation Perl. Vous pouvez lire cette documentation avec la commande `perldoc` ou toute autre méthode, par exemple celle que vous utilisez pour lire le présent document.

3.1.1 Qu'est-ce que Perl ?

Perl est un langage de programmation généraliste créé à l'origine pour la manipulation automatique de textes et désormais utilisé dans une large gamme de tâches, dont l'administration système, le développement web, la programmation réseau, la création d'interfaces graphiques et bien plus encore.

Le langage a pour but premier d'être pratique (facile à utiliser, efficace, complet) plutôt que beau (compact, élégant, minimaliste). Ses caractéristiques principales sont sa facilité d'utilisation, le support du style de programmation impératif (à base de procédures) et du style orienté objet (OO), l'intégration de puissantes capacités de manipulation de textes, et enfin l'une des plus impressionnantes collections au monde de modules complémentaires.

Diverses définitions de Perl peuvent être trouvées dans *perl*, *perlfaq1* et sans doute à bien d'autres endroits. Nous pouvons en déduire que différents groupes d'utilisateurs voient en Perl bien des choses différentes, mais surtout que pas mal de monde considère le sujet comme suffisamment intéressant pour en parler.

3.1.2 Exécuter des programmes Perl

Pour exécuter un programme Perl depuis la ligne de commande Unix :

```
perl progname.pl
```

Ou bien, placez la ligne suivante comme première ligne de votre script :

```
#!/usr/bin/env perl
```

...et exécutez le script en tapant `/dossier/contenant/le/script.pl`. Bien sûr, il faut d'abord que vous l'ayez rendu exécutable, en tapant `chmod 755 script.pl` (sous Unix).

(Cette première ligne suppose que vous disposez du programme `env`. Vous pouvez plutôt y placer directement le chemin d'accès de votre exécutable `perl`. Par exemple `#!/usr/bin/perl`.)

Pour plus d'informations, en particulier les instructions pour d'autres plates-formes comme Windows et Mac OS, référez vous à *perlrun*.

3.1.3 Filet de sécurité

Perl, par défaut, est très permissif. Afin de le rendre exigeant et donc plus robuste, il est recommandé de débiter chaque programme par les lignes suivantes :

```
#!/usr/bin/perl
use strict;
use warnings;
```

Ces deux lignes supplémentaires demandent à perl de rechercher dans votre code différents problèmes courants. Chacune d'entre elles permettant la vérification de choses différentes, vous avez donc besoin de ces deux lignes. La détection d'un problème potentiel par `use strict`; arrêtera votre code immédiatement. À l'inverse, `use warnings`; ne fera qu'afficher des avertissements (comme l'option `-w` de la ligne de commande) et laissera votre code continuer. Pour en savoir plus sur ces vérifications, lisez leur documentation respective : *strict* et *warnings*.

3.1.4 Les bases de la syntaxe

Un programme ou un script Perl est constitué d'une suite de "phrases". Ces phrases sont simplement saisies dans le script les unes à la suite des autres dans l'ordre désiré de leur exécution, de la manière la plus directe possible. Inutile de créer une fonction `main()` ou quoi que ce soit de la sorte.

Principale différence avec les phrases françaises, une phrase Perl se termine par un point-virgule :

```
print "Salut, la terre";
```

Les commentaires débutent par un symbole dièse et vont jusqu'à la fin de la ligne.

```
# Ceci est un commentaire
```

Les espaces ou passages à la ligne sont ignorés :

```
print
    "Salut, la terre"
;
```

...sauf à l'intérieur des chaînes de caractères entre guillemets :

```
# La phrase suivante passe à la ligne en plein milieu du texte
print "Salut
la terre";
```

Les chaînes de caractères littérales peuvent être délimitées par des guillemets doubles ou simples (apostrophes) :

```
print "Salut, la terre";
print 'Salut, la terre';
```

Il y a cependant une différence importante entre guillemets simple et doubles. Une chaîne de caractères encadrée par des apostrophes s'affiche telle qu'elle a été saisie, tandis que dans les chaînes entre guillemets doubles Perl remplace les variables et les caractères spéciaux (par exemple les fins de lignes `\n`) par leur valeur. On parle "d'interpolation" des variables.

```
$nom = "toto";
print "Bonjour, $nom\n";      # Affiche : Bonjour, toto
print 'Bonjour, $nom\n';     # Affiche littéralement : Bonjour, $name\n
```

Les nombres n'ont pas besoin d'être encadrés par des guillemets :

```
print 42;
```

Vous pouvez utiliser ou non des parenthèses pour les arguments des fonctions selon vos goûts personnels. Elles ne sont nécessaires que de manière occasionnelle pour clarifier des problèmes de priorité d'opérateur.

```
print("Salut, la terre\n");
print "Salut, la terre\n";
```

Vous trouverez des informations plus détaillées sur la syntaxe de Perl dans *perlsyn*.

3.1.5 Types de variables Perl

Perl propose trois types de variables : les scalaires, les tableaux et les tables de hachage.

Scalaires

Un scalaire représente une valeur unique :

```
my $animal = "chameau";
my $reponse = 42;
```

Les scalaires peuvent être indifféremment des chaînes de caractères, des entiers, des nombres en virgules flottante, des références (voir plus loin), plus quelques valeurs spéciales. Perl procédera automatiquement aux conversions nécessaires lorsque c'est cohérent (par exemple pour transformer en nombre une chaîne de caractère représentant un nombre). Il est inutile de déclarer au préalable le type des variables mais, en revanche, vous devez déclarer vos variables lors de leur première utilisation en utilisant le mot-clé `my`. (Ceci est l'une des exigences de `use strict`;))

Les valeurs scalaires peuvent être utilisées de différentes manières :

```
my $animal = "chameau";
print $animal;
print "L'animal est un $animal\n";
my $nombre = 42;
print "Le carré de $nombre est " . $nombre*$nombre . "\n";
```

Perl définit également la valeur scalaire `undef` (valeur d'une variable dans laquelle on n'a encore rien rangé), qu'on peut généralement considérer comme équivalente à une chaîne vide.

En Perl, en plus des variables définies dans le programme il existe un certain nombre de scalaires "magiques" qui ressemblent à des erreurs de ponctuation. Ces variables spéciales sont utilisées pour toutes sortes de choses. Elles sont documentées dans *perlvar*. Pour l'instant la seule dont vous devez vous préoccuper est la variable `$_`. On l'appelle la variable "par défaut". Elle est utilisée comme argument par défaut par quantité de fonctions Perl et modifiée implicitement par certaines structures de boucle.

```
print;      # Affiche, par défaut, le contenu de $_
```

Les tableaux

Un tableau (ou liste, il y a une différence mais dans la suite nous emploierons indifféremment les deux) représente une liste de valeurs :

```
my @animaux = ("chameau", "lama", "hibou");
my @nombres = (23, 42, 69);
my @melange = ("chameau", 42, 1.23);
```

Le premier élément d'un tableau se trouve à la position 0. Voici comment faire pour accéder aux éléments d'un tableau :

```
print $animaux[0];      # Affiche "chameau"
print $animaux[1];      # Affiche "lama"
```

La variable spéciale `$#tableau` donne l'index du dernier élément d'un tableau (c'est à dire 1 de moins que le nombre d'éléments puisque le tableau commence à zéro) :

```
print $melange[$#melange];      # dernier élément, affiche 1.23
```

Vous pourriez être tenté d'utiliser `$#array + 1` pour connaître le nombre d'éléments contenus dans un tableau... mais il y a plus simple. Il suffit en fait d'utiliser `@array` à un emplacement où Perl attend une valeur scalaire ("dans un contexte scalaire") et vous obtenez directement le nombre d'éléments du tableau.

```
if (@animaux < 5) { ... }
```

Les lecteurs attentifs auront sans doute remarqués que les éléments du tableau auxquels nous accédons commencent par `$`. L'idée est qu'en accédant à un élément nous demandons de retirer une seule valeur du tableau, on demande un scalaire, on obtient donc un scalaire.

Extraire simultanément plusieurs valeurs d'un tableau :

```
@animaux[0,1];          # donne ("chameau", "lama");
@animaux[0..2];         # donne ("chameau", "lama", "Hibou");
@animaux[1..$#animaux]; # donne tous les éléments sauf le premier
```

C'est ce qu'on appelle une tranche de tableau.

Vous pouvez réaliser quantité de choses utiles en manipulant des listes :

```
my @tri = sort @animaux; # tri
my @sansdessusdessous = reverse @nombres; # inversion
```

Comme pour les scalaires il existe quelques "tableaux magiques". Par exemple @ARGV (les arguments d'appel de votre script en ligne de commande) et @_ (les arguments transmis à un sous programme). Les tableaux spéciaux sont documentés dans *perlvar*.

Les tables de hachage

Une table de hachage (ou plus brièvement "un hash") représente un ensemble de paires clé/valeur :

```
my %fruit_couleur = ("pomme", "rouge", "banane", "jaune");
```

Vous pouvez utiliser des espaces et l'opérateur => pour présenter plus joliment le code précédent. => est un synonyme de la virgule possédant quelques propriétés particulière, par exemple se passer des guillemets pour encadrer le mot qui le précède. Cela donne :

```
my %fruit_couleur = (
    pomme => "rouge",
    banane => "jaune",
);
```

Pour accéder aux éléments d'un hash :

```
$fruit_couleur{"pomme"};          # donne "rouge"
```

Vous avez aussi la possibilité d'obtenir la liste de toutes les clés ou de toutes les valeurs du hash grâce aux fonctions `keys()` et `values()`.

```
my @fruits = keys %fruit_couleur;
my @couleurs = values %fruit_couleur;
```

Les tables de hachage n'ont pas d'ordre interne spécifique. Vous ne pouvez donc pas prévoir dans quel ordre les listes `keys()` et `values()` renverront leurs éléments (par contre vous pouvez être sûr qu'il s'agit du même ordre pour les deux). Si vous désirez accéder aux éléments d'un hash de manière triée, vous avez toujours la possibilité de trier la liste `keys` et d'utiliser une boucle pour parcourir les éléments.

Vous l'avez deviné : comme pour les scalaires et les tableaux, il existe également quelques "hashs" spéciaux. Le plus connu est %ENV qui contient les variables d'environnement du système. Pour tout savoir à son sujet (et sur les autres variables spéciales), regardez *perlvar*.

Les scalaires, les tableaux et les hashes sont documentés de manière plus complète dans *perlldata*.

Il est possible de construire des types de données plus complexes en utilisant les références qui permettent, par exemple, de construire des listes et des hashes à l'intérieur de listes et de hashes.

Une référence est une valeur scalaire qui (comme son nom l'indique) se réfère à n'importe quel autre type de données Perl. Donc en conservant des références en tant qu'élément d'un tableau ou d'un hash, on peut facilement créer des listes et des hash à l'intérieur de listes et de hashes. Pas de panique, c'est plus compliqué à décrire qu'à créer. L'exemple suivant montre comment construire un hash à deux niveaux à l'aide de références anonymes vers des hash.

```
my $variables = {
    scalaire => {
        description => "élément isolé",
        prefix => '$',
    },
    tableau => {
        description => "liste ordonnée d'éléments",
        prefix => '@',
    },
    hash => {
        description => "paire clé/valeur",
        prefix => '%',
    },
};

print "Les scalaires commencent par $variables->{'scalaire'}->{'prefix'}\n";
```

Des informations exhaustives sur les références peuvent être trouvées dans *perlrefut*, *perllol*, *perlref* et *perldsc*.

3.1.6 Portée des variables

Dans tout ce qui précède, pour définir des variables nous avons utilisé sans l'expliquer la syntaxe :

```
my $var = "valeur";
```

Le mot clé `my` est en réalité optionnel. Vous pourriez vous contenter d'écrire :

```
$var = "valeur";
```

Toutefois, la seconde version va créer des variables globales connues dans tout le programme, ce qui est une mauvaise pratique de programmation. Le mot clé `my` crée des variables à portées lexicales, c'est-à-dire qui ne sont connues qu'au sein du bloc (c.-à-d. un paquet de phrases entouré par des accolades) dans lequel elles sont définies.

```
my $x = "foo";
ly $une_condition = 1m
if ($une_condition) {
    my $y = "bar";
    print $x;           # Affiche "foo"
    print $y;           # Affiche "bar"
}
print $x;               # Affiche "foo"
print $y;               # N'affiche rien; $y est hors de portée
```

De plus, en utilisant `my` en combinaison avec la phrase `use strict;` au début de vos scripts Perl, l'interpréteur détectera un certain nombre d'erreurs de programmation communes. Dans l'exemple précédant, la phrase finale `print $b;` provoquerait une erreur de compilation et vous interdirait d'exécuter le programme (ce qui est souhaitable car quand il y a une erreur il vaut mieux la détecter le plus tôt possible). Utiliser `strict` est très fortement recommandé.

3.1.7 Structures conditionnelles et boucles

Perl dispose des structures d'exécution conditionnelles et des boucles usuelles des autres langages de programmation, à l'exception de `case/switch` (mais si vous y tenez vraiment il existe un module `Switch` intégré à partir de Perl 5.8 et disponible sur CPAN). Lisez ci-dessous la section consacrée aux modules pour plus d'informations sur les modules et CPAN).

Une condition peut être n'importe quelle expression Perl. Elle est considérée comme vraie ou fausse suivant sa valeur, 0 ou chaîne vide signifiant FAUX et toute autre valeur signifiant VRAIE. Voyez la liste des opérateurs dans la prochaine section pour savoir quels opérateurs logiques et booléens sont habituellement utilisés dans les expressions conditionnelles.

if

```
if ( condition ) {
    ...
} elsif ( autre condition ) {
    ...
} else {
    ...
}
```

Il existe également une version négative du `if` :

```
unless ( condition ) {
    ...
}
```

`unless` permet ainsi de disposer d'une version plus lisible de `if (!condition)`.

Notez que, contrairement aux pratiques d'autres langages, les accolades sont obligatoires en Perl même si le bloc conditionnel est réduit à une ligne. Il existe cependant un truc permettant de donner aux expressions conditionnelles d'une ligne un aspect plus proche de l'anglais courant :

```
# comme d'habitude
if ($zippy) {
    print "Yahou!";
}
unless ($bananes) {
    print "Y'a plus de bananes";
}

# la post-condition Perlienne
print "Yahou!" if $zippy;
print "Y'a plus de bananes" unless $bananes;
```

while

```
while ( condition ) {
    ...
}
```

Il existe également une version négative de `while` :

```
until ( condition ) {
    ...
}
```

Vous pouvez aussi utiliser `while` dans une post-condition :

```
print "LA LA LA\n" while 1;          # boucle infinie
```

for

La construction `for` fonctionne exactement comme en C :

```
for ($i = 0; $i <= $max; $i++) {
    ...
}
```

La boucle `for` à la mode C est toutefois rarement nécessaire en Perl dans la mesure où Perl fournit une alternative plus intuitive : la boucle de parcours de liste `foreach`.

foreach

```
foreach (@array) {
    print "L'élément courant est $_\n";
}

print $list[$_] foreach 0 .. $max;

# vous n'êtes pas non plus obligé d'utiliser $_ ...
foreach my $cle (keys %hash) {
    print "La valeur de $cle est $hash{$cle}\n";
}
```

En pratique on peut substituer librement `for` à `foreach` et inversement. Perl se charge de détecter la variante utilisée.

Pour plus de détails sur les boucles (ainsi qu'un certain nombre de choses que nous n'avons pas mentionnées ici) consultez *perlsyn*.

3.1.8 Opérateurs et fonctions internes

Perl dispose d'une large gamme de fonctions internes. Nous avons déjà vu quelques unes d'entre elles dans les exemples précédents : `print`, `sort` et `reverse`. Pour avoir une liste des fonctions disponibles consultez *perlfunc*. Vous pouvez facilement accéder à la documentation de n'importe quelle fonction en utilisant `perldoc -f fonctionname`.

Les opérateurs Perl sont entièrement documentés dans *perlop*. Voici déjà quelques-uns parmi les plus utilisés :

Arithmétique

```
+   addition
-   soustraction
*   multiplication
/   division
```

Comparaison numérique


```

== égalité
!= inégalité
< inférieur
> supérieur
<= inférieur ou égal
>= supérieur ou égal

```

Comparaison de chaînes

```

eq égalité
ne inégalité
lt inférieur
gt supérieur
le inférieur ou égal
ge supérieur ou égal

```

Pourquoi Perl propose-t-il des opérateurs différents pour les comparaisons numériques et les comparaisons de chaînes ? Parce qu'en Perl il n'existe pas de différence de type entre variables numériques ou chaînes de caractères. Perl a donc besoin de savoir s'il faut comparer les éléments suivants dans l'ordre numérique (ou 99 est inférieur à 100) ou dans l'ordre alphabétique (où 100 vient avant 99).

Logique booléenne

```

&& and          et
|| or           ou
! not          négation

```

and, or et not ne sont pas mentionnés dans la table uniquement en tant que description des opérateurs symboliques correspondants – ils existent comme opérateurs en tant que tels. Leur raison d'être n'est pas uniquement d'offrir une meilleure lisibilité que leurs équivalents C. Ils ont surtout une priorité différente de && et consorts. Lisez *perlop* pour de plus amples détails.

Divers

```

= affectation
. concaténation de chaînes
x multiplication de chaînes
.. opérateur d'intervalle (crée une liste de nombres)

```

De nombreux opérateurs peuvent être combinés avec un =

```

$a += 1;      # comme $a = $a + 1
$a -= 1;      # comme $a = $a - 1
$a .= "\n";   # comme $a = $a . "\n";

```

3.1.9 Fichiers et E/S (entrées/sorties)

Vous pouvez ouvrir un fichier en entrée ou en sortie grâce à la fonction `open()`. Celle-ci est documentée avec un luxe extravagant de détails dans *perlfunc* et *perlopentut*. Plus brièvement :

```

open(my $in, "<", "input.txt")
  or die "Impossible d'ouvrir input.txt en lecture : $!";
open(my $out, ">", "output.txt")
  or die "Impossible d'ouvrir output.txt en écriture : $!";
open(my $log, ">>", "my.log")
  or die "Impossible d'ouvrir my.log en ajout : $!";

```

Pour lire depuis un descripteur de fichier ouvert on utilise l'opérateur `<>`. Dans un contexte scalaire, cet opérateur lit une ligne du fichier associé. Dans un contexte de liste, il lit l'intégralité du fichier en rangeant chaque ligne dans un élément de la liste.

```

my $ligne = <$in>;
my @lignes = <în>;

```

Lire un fichier entier en une seule fois se dit "slurper". Même si cela peut parfois s'avérer utile, c'est généralement un gâchis de mémoire. La majorité des traitements de nécessité pas de lire plus d'une ligne à la fois en utilisant les structures de boucles de Perl.

L'opérateur `<>` apparaît en général dans une boucle `while` :

```
while (<$in>) { # chaque ligne est successivement affectée à $_
    print "Je viens de lire la ligne : $_";
}
```

Nous avons déjà vu comment écrire sur la sortie standard en utilisant la fonction `print()`. Celle-ci peut également prendre comme premier argument optionnel un descripteur de fichier, précisant dans quel fichier l'écriture doit avoir lieu :

```
print STDERR "Dernier avertissement.\n";
print $out $record;
print $log $logmessage;
```

Quand vous avez fini de travailler avec vos descripteurs de fichier, vous devez en principe les fermer à l'aide de la fonction `close()` (quoique pour être tout à fait honnêtes, Perl se chargera de faire le ménage si vous oubliez) :

```
close $in;
```

3.1.10 Expressions régulières (ou rationnelles)

Dans le jargon informatique on désigne par "expressions régulières" (ou rationnelles) une syntaxe utilisée pour définir des motifs recherchés dans un texte ou une chaîne de caractères. Le support par Perl des expressions régulières est à la fois large et puissant et c'est le sujet d'une documentation très complète : *perlrequick*, *perlretut* et autres. Nous allons les présenter brièvement :

Détection de motifs simples

```
if (/foo/) { ... } # vrai si $_ contient "foo"
if ($a =~ /foo/) { ... } # vrai si $a contient "foo"
```

L'opérateur `//` de détection de motif est documenté dans *perlop*. Par défaut il travaille sur la variable `$_` ou peut être appliqué à une autre variable en utilisant l'opérateur de liaison `=~` (lui aussi documenté dans *perlop*).

Substitution simple

```
s/foo/bar/; # remplace foo par bar dans $_
$a =~ s/foo/bar/; # remplace foo par bar dans $a
$a =~ s/foo/bar/g; # remplace TOUTES LES INSTANCES de foo par bar dans $a
```

L'opérateur de substitution `s///` est documenté à la page *perlop*.

Expressions régulières plus complexes

Vous n'êtes pas limité à la détection de motifs fixes (si c'était le cas, on ne parlerait d'ailleurs pas d'expressions régulières, *regexp* pour les intimes). En pratique il est possible de détecter pratiquement n'importe quel motif imaginable en utilisant des expressions régulières plus complexes. Celles-ci sont documentées en profondeur dans *perlre*. Pour vous mettre en bouche voici déjà une petite antisèche :

.	un caractère unique (n'importe lequel)
\s	un blanc (espace, tabulation, passage à la ligne...)
\S	un caractère non-blanc (le contraire du précédent)
\d	un chiffre (0-9)
\D	un non-chiffre
\w	un caractère alphanumérique (a-z, A-Z, 0-9, _)
\W	un non-alphanumérique
[aeiou]	n'importe quel caractère de l'ensemble entre crochets
[^aeiou]	n'importe quel caractère sauf ceux de l'ensemble entre crochets
(foo bar baz)	n'importe laquelle des alternatives proposées
^	le début d'une chaîne de caractères
\$	la fin d'une chaîne de caractères

Des quantificateurs peuvent être utilisés pour indiquer combien des éléments précédents vous désirez, un élément désignant aussi bien un caractère littéral qu'un des méta-caractères énumérés plus haut, ou encore un groupe de caractères ou de méta-caractères entre parenthèses.

*	zéro ou plus
+	un ou plus
?	zéro ou un
{3}	exactement 3 fois l'élément précédent
{3,6}	entre 3 et 6 fois l'élément précédent
{3,}	3 ou plus des éléments précédents

Quelques exemples rapides :

```

/^d+/      une chaîne commençant par un chiffre ou plus
/^$/      une chaîne vide (le début et la fin sont adjacents)
/(\d\s){3}/ un groupe de trois chiffres, chacun suivi par un blanc
           (par exemple "3 4 5 ")
/(a.+)/   une chaîne dont toutes les lettres impaires sont des a
           (par exemple "abacadaf")

# La boucle suivante lit l'entrée standard
# et affiche toutes les lignes non vides :
while (<>) {
    next if /^$/;
    print;
}

```

Capturer grâce aux parenthèses

En plus de créer un regroupement de caractères sur lequel utilisé un quantificateur, les parenthèses servent un second but. Elles peuvent être utilisées pour capturer les résultats d'une portion de regexp pour un usage ultérieur. Les résultats sont conservés dans les variables \$1, \$2 et ainsi de suite.

```

# la méthode du pauvre pour décomposer une adresse e-mail
if ($email =~ /([^\@]+)@(.+)/) {
    print "Compte : $1\n";
    print "Hôte   : $2\n";
}

```

Autres possibilités des regexp

Les regexp de Perl supportent également les références arrière, les références avant et toutes sortes d'autres constructions complexes. Pour tout savoir lisez *perlrequick*, *perlretut* et *perlre*.

3.1.11 Écriture de sous-programmes

Rien n'est plus facile que de déclarer un sous-programme :

```

sub logger {
    my $logmessage = shift;
    open my $logfile, ">>", "my.log" or die "Impossible d'ouvrir my.log: $!";
    print $logfile $logmessage;
}

```

Vous pouvez maintenant utiliser ce sous-programme comme n'importe quelle autre fonction prédéfinie :

```
logger("Nous avons un sous-programme de log !");
```

Que peut bien vouloir dire ce `shift` ? En réalité, comme nous l'avions évoqué plus haut, les paramètres sont transmis aux sous-programmes à travers le tableau magique `@_` (voir *perlvar* pour plus de détails). Il se trouve que la fonction `shift`, qui attend une liste, utilise `@_` comme argument par défaut. Donc la ligne `my $logmessage = shift;` extrait le premier argument de la liste et le range dans `$logmessage`.

Il existe d'autres façons de manipuler `@_` :

```

my ($logmessage, $priority) = @_;      # fréquent
my $logmessage = $_[0];                # plus rare

```

Les sous-programmes peuvent bien entendu retourner des résultats :

```

sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}

```

Utilisez-le comme ceci :

```
$sq = square(8);
```

Le sujet est évidemment beaucoup plus complexe. Pour plus d'informations sur l'écriture de sous-programmes, voyez *perlsub*.

3.1.12 Perl orienté objet

Les possibilités objet de Perl sont relativement simples. Elles sont implémentées en utilisant des références qui connaissent le type d'objet sur lesquelles elles pointent en se basant sur le concept de paquetage de Perl. La programmation objet en Perl dépasse largement le cadre de ce document. Lisez plutôt *perlboot*, *perltoot*, *perlooc* et *perlobj*.

En tant que débutant en Perl, vous utiliserez sans doute rapidement des modules tierces-parties, dont l'utilisation est brièvement décrite ci-dessous.

3.1.13 Utilisations de modules Perl

Les modules Perl fournissent quantité de fonctionnalités qui vous éviteront de réinventer sans cesse la roue. Il suffit de les télécharger depuis le site CPAN (<http://www.cpan.org/>). Beaucoup de modules sont aussi inclus dans la distribution Perl elle-même.

Les catégories de modules vont de la manipulation de texte aux protocoles réseau, en passant par l'accès aux bases de données ou au graphisme. CPAN présente la liste des modules classés par catégorie.

Pour apprendre à installer les modules téléchargés depuis CPAN, lisez *perlmodinstall*.

Pour apprendre à utiliser un module particulier, utilisez `perldoc Module::Name`. Typiquement vous commencerez par un `use Module::Name`, qui vous donnera accès aux fonctions exportées ou à l'interface orientée objet du module.

perlfaq contient une liste de questions et les solutions à de nombreux problèmes communs en Perl et propose souvent de bons modules CPAN à utiliser.

perlmod décrit les modules Perl modules de manière plus générale. *perlmodlib* énumère les modules qui accompagnent votre installation Perl.

Si écrire des modules Perl vous démange, *perlnewmod* vous donnera d'excellents conseils.

3.2 AUTEUR

Kirrily "Skud" Robert <skud@cpan.org>.

L'utilisation de l'image du chameau en association avec le langage Perl est une marque déposée de O'Reilly & Associates (<http://www.oreilly.com/>). Utilisé avec permission.

3.3 TRADUCTION

3.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

3.3.2 Traducteur

Pour la traduction initiale, Christophe Grosjean <krisssg@wanadoo.fr> et pour la mise à jour en version 5.10.0, Paul Gaborit (Paul.Gaborit at enstimac.fr).

3.3.3 Relecture

Paul Gaborit (Paul.Gaborit at enstimac.fr).

Deuxième partie

Tutoriels

Chapitre 4

perlreftut

Le très court tutoriel de Mark sur les références

4.1 DESCRIPTION

Une des caractéristiques nouvelles les plus importantes de Perl 5 a été la capacité de gérer des structures de données compliquées comme les tableaux multidimensionnels et les tables de hachage imbriquées. Pour les rendre possibles, Perl 5 a introduit un mécanisme appelé "références" ; l'utilisation de celles-ci est le moyen de travailler avec des données complexes structurées en Perl. Malheureusement, cela fait une grande quantité de syntaxe nouvelle à apprendre, et la page de manuel est parfois difficile à suivre. Cette dernière est très complète, et cela peut être un problème parce qu'il est difficile d'en tirer ce qui est important et ce qui ne l'est pas.

Heureusement, vous avez seulement besoin de savoir 10% de ce qui est dans la page de manuel principale pour tirer 90% de son bénéfice. Cette page espère vous montrer ces 10%.

4.2 Qui a besoin de structures de données compliquées ?

Un problème qui revenait souvent avec Perl 4 était celui de la représentation d'une table de hachage dont les valeurs sont des listes. Perl 4 avait les tables de hachage, bien sûr, mais les valeurs devaient être des scalaires ; elles ne pouvaient être des listes.

Pourquoi voudrait-on utiliser une table de hachage contenant des listes ? Prenons un exemple simple. Vous avez un fichier contenant des noms de villes et de pays, comme ceci :

```
Chicago, USA
Frankfort, Allemagne
Berlin, Allemagne
Washington, USA
Helsinki, Finlande
New York, USA
```

et vous voulez produire une sortie comme cela, avec chaque pays mentionné une seule fois, suivi d'une liste des villes de ce pays :

```
Finlande: Helsinki.
Allemagne: Berlin, Frankfort.
USA: Chicago, New York, Washington.
```

La façon naturelle de faire ceci est de construire une table de hachage dont les clés sont les noms des pays. A chaque pays clé on associe une liste des villes de ce pays. Chaque fois qu'on lit une ligne en entrée, on la sépare en un pays et une ville, on recherche la liste de villes déjà connues pour ce pays, et on y ajoute la nouvelle ville. Quand on a fini de lire les données en entrée, on parcourt la table de hachage, en triant chaque liste de villes avant de l'afficher.

Si les valeurs des tables de hachage ne peuvent être des listes, c'est perdu. En Perl 4, c'est le cas ; ces valeurs ne peuvent être que de chaînes de caractères. C'est donc perdu. Nous devrions probablement essayer de combiner toutes les villes dans une seule grande chaîne de caractères, et au moment d'écrire la sortie, couper cette chaîne pour en faire une liste, la trier, et la reconvertir en chaîne de caractères. C'est compliqué, et il est facile de faire des erreurs dans le processus. C'est surtout frustrant, parce que Perl a déjà de très belles listes qui résoudreiraient le problème, si seulement nous pouvions les utiliser.

4.3 La solution

Avant même que Perl 5 ne pointât le bout de son nez, nous étions donc coincés avec ce fait de conception : les valeurs de tables de hachage doivent être des scalaires. Les références sont la solution à ce problème.

Une référence est une valeur scalaire qui *fait référence* à un tableau entier ou à une table de hachage entière (ou à à peu près n'importe quoi d'autre). Les noms sont une forme de référence dont vous êtes déjà familier. Pensez au Président des États-Unis D'Amérique: ce n'est qu'un tas désordonné et inutilisable de sang et d'os. Mais pour parler de lui, ou le représenter dans un programme d'ordinateur, tout ce dont vous avez besoin est le sympathique et maniable scalaire "George Bush".

Les références en Perl sont comme des noms pour les tableaux et les tables de hachage. Ce sont des noms privés et internes de Perl, vous pouvez donc être sûr qu'ils ne sont pas ambigus. Au contraire de "George Bush", une référence pointe vers une seule chose, et il est toujours possible de savoir vers quoi. Si vous avez une référence à un tableau, vous pouvez accéder au tableau complet qui est derrière. Si vous avez une référence à une table de hachage, vous pouvez accéder à la table entière. Mais la référence reste un scalaire, compact et facile à manipuler.

Vous ne pouvez pas construire une table de hachage dont les valeurs sont des tableaux : les valeurs des tables de hachage ne peuvent être que des scalaires. Nous y sommes condamnés. Mais une seule référence peut pointer vers un tableau entier, et les références sont des scalaires, donc vous pouvez avoir une table de hachage de références à des tableaux, et elle agira presque comme une table de hachage de tableaux, et elle pourra servir juste comme une table de hachage de tableaux.

Nous reviendrons à notre problème de villes et de pays plus tard, après avoir introduit un peu de syntaxe pour gérer les références.

4.4 Syntaxe

Il n'y a que deux façons de construire une référence, et deux façons de l'utiliser une fois qu'on l'a.

4.4.1 Construire des références

Règle de construction 1

Si vous mettez un \ devant une variable, vous obtenez une référence à cette variable.

```
$tref = \@tableau;      # $tref contient maintenant une référence
                        # à @tableau
$href = \%hachage;     # $href contient maintenant une référence
                        # à %hachage
$sref = \$scalaire;    # $sref contient maintenant une référence
                        # à $scalaire
```

Une fois que la référence est stockée dans une variable comme \$tref ou \$href, vous pouvez la copier ou la stocker ailleurs comme n'importe quelle autre valeur scalaire :

```
$xy = $tref;           # $xy contient maintenant une référence
                        # à @tableau
$p[3] = $href;        # $p[3] contient maintenant une référence
                        # à %hachage
$z = $p[3];           # $z contient maintenant une référence
                        # à %hachage
```

Ces exemples montrent comment créer des références à des variables avec un nom. Parfois, on veut utiliser un tableau ou une table de hachage qui n'a pas de nom. C'est un peu comme pour les chaînes de caractères et les nombres : on veut être capable d'utiliser la chaîne "\n" ou le nombre 80 sans avoir à les stocker dans une variable nommée d'abord.

Règle de construction 2

[ELEMENTS] crée un nouveau tableau anonyme, et renvoie une référence à ce tableau. { ELEMENTS } crée une nouvelle table de hachage anonyme et renvoie une référence à cette table.

```
$tref = [ 1, "toto", undef, 13 ];
# $tref contient maintenant une référence à un tableau

$href = { AVR => 4, AOU => 8 };
# $href contient maintenant une référence à une table de hachage
```

Les références obtenues grâce à la règle 2 sont de la même espèce que celles obtenues par la règle 1 :

```
# Ceci :
$tref = [ 1, 2, 3 ];

# Fait la même chose que cela :
@tableau = (1, 2, 3);
$tref = \@tableau;
```

La première ligne est une abréviation des deux lignes suivantes, à ceci près qu'elle ne crée pas la variable tableau superflue @tableau.

Si vous écrivez [], vous obtenez une référence à un nouveau tableau anonyme vide. Si vous écrivez {}, vous obtenez une référence à un nouvelle table de hachage anonyme vide.

4.4.2 Utiliser les références

Une fois qu'on a une référence, que peut-on en faire? C'est une valeur scalaire, et nous avons vu que nous pouvons la stocker et la récupérer ensuite comme n'importe quel autre scalaire. Il y a juste deux façons de plus de l'utiliser :

Règle d'utilisation 1

À la place du nom d'un tableau, vous pouvez toujours utiliser une référence à un tableau, tout simplement en la plaçant entre accolades. Par exemple, @{\$tref} la place de @tableau.

En voici quelques exemples :

Tableaux :

@t	@{\$tref}	Un tableau
reverse @t	reverse @{\$tref}	Inverser un tableau
\$t[3]	\${tref}[3]	Un élément du tableau
\$t[3] = 17;	\${tref}[3] = 17	Affecter un élément

Sur chaque ligne, les deux expressions font la même chose. Celles de gauche travaillent sur le tableau @t, et celles de droite travaillent sur le tableau vers lequel pointe \$tref; mais une fois qu'elle ont accédé au tableau sur lequel elles opèrent, elles leur font exactement la même chose.

On utilise une référence à une table de hachage exactement de la même façon :

%h	%{\$href}	Une table de hachage
keys %h	keys %{\$href}	Obtenir les clés de la table
\$h{'bleu'}	\${href}{'bleu'}	Un élément de la table
\$h{'bleu'} = 17	\${href}{'bleu'} = 17	Affecter un élément

Quelque soit ce que vous voulez faire avec une référence, vous pouvez y arriver en appliquant la **Règle d'utilisation 1**. Vous commencez par écrire votre code Perl en utilisant un vrai tableau ou une vraie table de hachage puis vous remplacez le nom du tableau ou de la table par {\$reference}.

"Comment faire une boucle sur un tableau pour lequel je ne possède qu'une référence?" Pour faire une boucle sur un tableau "normal" vous auriez écrit :


```
for my $element (@tableau) {
    ...
}
```

Remplaçons donc le nom du tableau (tableau) par la référence :

```
for my $element (@{$tref}) {
    ...
}
```

"Comment afficher le contenu d'une table de hachage dont je possède une référence ?" Tout d'abord, écrivons le code pour afficher le contenu d'une table de hachage :

```
for my $cle (keys %hachage) {
    print "$cle => $hachage{$cle}\n";
}
```

Puis remplaçons le nom de la table par la référence :

```
for my $cle (keys %${href}) {
    print "$cle => ${href}{$cle}\n";
}
```

Règle d'utilisation 2

La **Règle d'utilisation 1** est la seule réellement indispensable puisqu'elle vous permet de faire tout ce que vous voulez avec des références. Mais le chose la plus courante que l'on fait avec un tableau ou une table de hachage est d'accéder à une valeur et la notation induite par la **Règle d'utilisation 1** est lourde. Il existe donc une notation raccourcie.

`${tref}[3]` est trop dur à lire, vous pouvez donc écrire `$tref->[3]` à la place.

`${href}{bleu}` est trop dur à lire, vous pouvez donc écrire `$href->{bleu}` à la place.

Si `$tref` est une référence à un tableau, alors `$tref->[3]` est le quatrième élément du tableau. Ne mélangez pas ça avec `$tref[3]`, qui est le quatrième élément d'un tout autre tableau, trompeusement nommé `@tref`. `$tref` et `@tref` ne sont pas reliés, pas plus que `$truc` et `@truc`.

De la même façon, `$href->{bleu}` est une partie de la table de hachage vers laquelle pointe `$href`, et qui ne porte peut-être pas de nom. `href{bleu}` est une partie de la table de hachage trompeusement nommée `%href`. Il est facile d'oublier le `->`, et si cela vous arrive vous obtiendrez des résultats étranges comme votre programme utilisera des tableaux et des tables de hachage issus de tableaux et de tables de hachages qui n'étaient pas ceux que vous aviez l'intention d'utiliser.

4.5 Un exemple

Voyons un rapide exemple de l'utilité de tout ceci.

D'abord, souvenez-vous que `[1, 2, 3]` construit un tableau anonyme contenant (1, 2, 3), et vous renvoie une référence à ce tableau.

Maintenant, considérez

```
@t = ( [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
      );
```

`@t` est un tableau à trois éléments, et chacun d'entre eux est une référence à un autre tableau.

`$t[1]` est l'une de ces références. Elle pointe vers un tableau, celui contenant (4, 5, 6), et puisque c'est une référence à un tableau, la Règle d'utilisation 2 dit que nous pouvons écrire `$t[1]->[2]` pour accéder au troisième élément du tableau. `$t[1]->[2]` vaut 6. De la même façon, `$t[0]->[1]` vaut 2. Nous avons ce qui ressemble à un tableau de dimensions deux ; vous pouvez écrire `$t[LIGNE]->[COLONNE]` pour obtenir ou modifier l'élément se trouvant à une ligne et une colonne données du tableau.

Notre notation est toujours quelque peu maladroite, voici donc un raccourci de plus :

4.6 Règle de la flèche

Entre deux **indices**, la flèche et facultative.

A la place de `$t[1]->[2]`, nous pouvons écrire `$t[1][2]` ; cela veut dire la même chose. A la place de `$t[0]->[1] = 23`, nous pouvons écrire `$t[0][1] = 23` ; cela veut dire la même chose.

Maintenant on dirait vraiment un tableau à deux dimensions !

Vous pouvez voir pourquoi les flèches sont importantes. Sans elles, nous devrions écrire `$$t[1][2]` à la place de `$t[1][2]`. Pour les tableaux à trois dimensions, elles nous permettent d'écrire `$x[2][3][5]` à la place de l'illisible `$$x[2][3][5]`.

4.7 Solution

Voici maintenant la réponse au problème que j'ai posé plus haut, qui consistait à reformater un fichier de villes et de pays.

```

1  my %table;
2  while (<>) {
3      chomp;
4      my ($ville, $pays) = split /, /;
5      $table{$pays} = [] unless exists $table{$pays};
6      push @{$table{$pays}}, $ville;
7  }

8  foreach $pays (sort keys %table) {
9      print "$pays: ";
10     my @villes = @{$table{$pays}};
11     print join ', ', sort @villes;
12     print ".\n";
13 }

```

Ce programme est constitué de deux parties : les lignes 2 à 7 lisent les données et construisent une structure de données puis les lignes 8 à 13 analysent les données et impriment un rapport. Nous allons obtenir une table de hachage dont les clés sont les noms de pays et dont les valeurs sont des références vers des tableaux de noms de villes. La structure de données ressemblera à cela :

```

%table
+-----+-----+
|      | | | +-----+-----+
|Germany| *---->| Frankfurt | Berlin |
|      | | | +-----+-----+
+-----+-----+
|      | | | +-----+
|Finland| *---->| Helsinki |
|      | | | +-----+
+-----+-----+
|      | | | +-----+-----+-----+
|  USA  | *---->| Chicago | Washington | New York |
|      | | | +-----+-----+-----+
+-----+-----+

```

Commençons par détailler la seconde partie du code. Supposons donc que nous avons déjà cette structure. Comment l'afficher ?

```

8  foreach $pays (sort keys %table) {
9      print "$pays: ";
10     my @villes = @{$table{$pays}};
11     print join ', ', sort @villes;
12     print ".\n";
13 }

```

`%table` est une table de hachage ordinaire dont on extrait les clés en les triant afin de les parcourir. La seule utilisation d'une référence est en ligne 10. `$table{$pays}` accède, via la clé `$pays`, à une valeur qui est une référence à un tableau de villes de ce pays. La **Règle d'utilisation 1** indique que nous pouvons obtenir ce tableau en écrivant `@{$table{$pays}}`. La ligne 10 est donc comme :

```
@villes = @tableau;
```

sauf que le nom `tableau` a été remplacé par la référence `{ $table{$pays} }`. Le `@` indique à Perl qu'on veut tout le tableau. Une fois le tableau des villes récupéré, on trie et on concatène ses valeurs pour finalement les afficher comme d'habitude.

Les ligne 2 à 7 fabriquent la structure de données. Les voici à nouveau :

```
2 while (<>) {
3     chomp;
4     my ($ville, $pays) = split /, /;
5     $table{$pays} = [] unless exists $table{$pays};
6     push @{$table{$pays}}, $ville;
7 }
```

Les lignes 2 à 4 récupèrent les noms d'une ville et d'un pays. La ligne 5 regarde si ce pays existe déjà en tant que clé de la table de hachage. Si elle n'existe pas, le programme utilise la notation `[]` (**Règle de construction 2**) pour créer une référence à un nouveau tableau anonyme vide prêt à accueillir des noms de villes. Cette référence est associée à la clé appropriée dans la table de hachage.

La ligne 6 ajoute le nom de la ville dans le tableau approprié. `$table{$pays}` contient une référence au tableau de noms de villes du pays concerné. La ligne 6 est exactement comme :

```
push @tableau, $ville;
```

sauf que le nom `tableau` a été remplacé par la référence `{ $table{$pays} }`. Le `push` ajoute un nom de ville à la fin du tableau référencé.

Il y a un détail que j'ai omis. La ligne 5 est en fait inutile et nous pourrions écrire :

```
2 while (<>) {
3     chomp;
4     my ($ville, $pays) = split /, /;
5     ##### $table{$pays} = [] unless exists $table{$pays};
6     push @{$table{$pays}}, $ville;
7 }
```

Si, dans `%table`, il y a déjà une valeur associée à la clé `$pays`, cela ne change rien. La ligne 6 retrouve la valeur de `$table{$pays}` qui est une référence à un tableau et ajoute la ville dans ce tableau. Mais que se passe-t-il si `$pays` contient un clé, disons Grèce, qui n'existe pas encore dans `%table` ?

Comme c'est du Perl, il se passe exactement ce qu'il faut. Perl voit que nous voulons ajouter Athènes à un tableau qui n'existe pas alors il crée un nouveau tableau vide pour nous, l'inscrit dans `%table` et y ajoute Athènes. Ceci s'appelle "l'autovivification" – donner vie à quelque chose automagiquement. Perl voit que la clé n'existe pas dans la table de hachage et donc il crée cette nouvelle clé automatiquement. Perl voit ensuite que vous voulez utiliser cette valeur comme une référence à un tableau alors il crée un nouveau tableau vide et stocke automatiquement sa référence dans la table de hachage. Ensuite, comme d'habitude, Perl augmente la taille du tableau afin d'y ajouter le nouveau nom.

4.8 Le reste

J'ai promis de vous donner accès à 90% du bénéfice avec 10% des détails, et cela implique que j'ai laissé de côté 90% des détails. Maintenant que vous avez une vue d'ensemble des éléments importants, il devrait vous être plus facile de lire la page de manuel *perlref*, qui passe en revue 100% des détails.

Quelques unes de grande lignes de *perlref* :

- Vous pouvez créer des références à n'importe quoi, ce qui inclut des scalaires, des fonctions, et d'autres références.

- Dans la **Règle d'utilisation 1**, vous pouvez omettre les accolades quand ce qui est à l'intérieur est une variable scalaire atomique comme `$tref`. Par exemple, `@$tref` est identique à `@{$tref}`, et `$$tref[1]` est identique à `{{$tref}[1]`. Si vous débutez, vous préférerez sûrement adopter l'habitude de toujours écrire les accolades.
- Le code suivant ne recopie pas le tableau référencé par `$tref1` :


```
$tref2 = $tref1;
```

 Vous obtenez en fait deux références au même tableau. Si vous modifiez `$tref1->[23]` et que vous regardez la valeur de `$tref2->[23]` vous verrez la modification.
 Pour copier le contenu du tableau, écrivez :


```
$tref2 = [@$tref1];
```

 On utilise la notation `[...]` pour créer un nouveau tableau anonyme et affecter sa référence à `$tref2`. Le contenu du nouveau tableau est initialisé avec les valeurs contenues dans le tableau référencé par `$tref1`.
 De manière similaire, pour copier un table de hachage anonyme, on écrit :


```
$href2 = {%{$href1}};
```
- Pour voir si une variable contient une référence, utilisez la fonction "ref". Elle renvoie vrai si son argument est une référence. En fait, c'est encore mieux que ça : elle renvoie HASH pour les références à une table de hachage et ARRAY pour les références à un tableau.
- Si vous essayez d'utiliser une référence comme une chaîne de caractères, vous obtenez quelque-chose comme


```
ARRAY(0x80f5dec) ou HASH(0x826afc0)
```

 Si jamais vous voyez une chaîne de caractères qui ressemble à ça, vous saurez que vous avez imprimé une référence par erreur.
 Un effet de bord de cette représentation est que vous pouvez utiliser `eq` pour savoir si deux références pointent vers la même chose. (Mais d'ordinaire vous devriez plutôt utiliser `==` parce que c'est beaucoup plus rapide.)
- Vous pouvez utiliser une chaîne de caractères comme si c'était une référence. Si vous utilisez la chaîne "foo" comme une référence à un tableau, elle est prise comme une référence au tableau `@foo`. On appelle cela une **référence symbolique**. Le pragma `use strict 'refs'` (NdT : et même `use strict`) désactive cette fonctionnalité qui, lorsqu'elle est utilisée accidentellement, peut provoquer plein de bugs.

Vous pourriez préférer continuer avec *perllol* au lieu de *perlref* : il y est traité des listes de listes et des tableaux multi-dimensionnels en détail. Après cela, vous devriez avancer vers *perldsc* : c'est un livre de recettes pour les structures de données (*Data Structure Cookbook*) qui montre les recettes pour utiliser et imprimer les tableaux de tables de hachage, les tables de hachage de tableaux, et les autres sortes de données.

4.9 Résumé

Tout le monde a besoin de structures de données complexes, et les références sont le moyen d'y accéder en Perl. Il y a quatre règles importantes pour manipuler les références : deux pour les construire, et deux pour les utiliser. Une fois que vous connaissez ces règles, vous pouvez faire la plupart des choses importantes que vous devez faire avec des références.

4.10 Crédits

Auteur : Mark Jason Dominus, Plover Systems (mjd-perl-ref+plover.com)

Cet article est paru à l'origine dans *The Perl Journal* (<http://www.tpj.com/>) volume 3, # 2. Réimprimé avec permission.

Le titre original était *Understand References Today (Comprendre les références aujourd'hui)*.

4.10.1 Distribution conditions

Copyright 1998 The Perl Journal (<http://www.tpj.com/>).

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

4.10.2 Conditions de distribution

Copyright 1998 The Perl Journal (<http://www.tpj.com/>).

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même.

Quel que soit leur mode de distribution, tous les exemples de code de ces fichiers sont par ce document placés dans le domaine public. Vous êtes autorisé et encouragé à utiliser ce code dans vos programmes à fins commerciales ou d'amusement, comme vous le souhaitez. Un simple commentaire dans le code signalant son origine serait courtois mais n'est pas requis.

4.11 TRADUCTION

4.11.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

4.11.2 Traducteur

Traduction initiale : Ronan Le Hy (rlehy@free.fr). Mise à jour : Paul Gaborit (paul.gaborit@enstimac.fr).

4.11.3 Relecture

Personne pour l'instant.

Chapitre 5

perldsc

Livre de recettes des structures de données en Perl

5.1 DESCRIPTION

La seule caractéristique manquant cruellement au langage de programmation Perl avant sa version 5.0 était les structures de données complexes. Même sans support direct par le langage, certains programmeurs vaillants parvinrent à les émuler mais c'était un dur travail, à déconseiller aux âmes sensibles. Vous pouviez occasionnellement vous en sortir avec la notation `$m{ $AoA, $b }` empruntée à *awk* dans laquelle les clés sont en fait une seule chaîne concaténée "`AoAb`", mais leurs parcours et leurs tris étaient difficiles. Des programmeurs un peu plus désespérés ont même directement bidouillé la table de symboles interne de Perl, une stratégie qui s'est montrée dure à développer et à maintenir - c'est le moins que l'on puisse dire.

La version 5.0 de Perl met à notre disposition des structures de données complexes. Vous pouvez maintenant écrire quelque chose comme ce qui suit pour obtenir d'un seul coup un tableau à trois dimensions !

```
for $x (1 .. 10) {
  for $y (1 .. 10) {
    for $z (1 .. 10) {
      $AoA[$x][$y][$z] =
        $x ** $y + $z;
    }
  }
}
```

À première vue cette construction apparaît simple, mais elle est en fait beaucoup plus compliquée qu'elle ne le laisse paraître !

Comment pouvez-vous l'imprimer ? Pourquoi ne pouvez-vous pas juste dire `print @AoA` ? Comment la triez-vous ? Comment pouvez-vous la passer à une fonction ou en récupérer une depuis une fonction ? Est-ce un objet ? Pouvez-vous la sauver sur disque pour la relire plus tard ? Comment accédez-vous à des lignes ou à des colonnes entières de cette matrice ? Est-ce que toutes les valeurs doivent être numériques ?

Comme vous le voyez, il est assez facile d'être déconcerté. Ces difficultés viennent, pour partie, de l'implémentation basée sur les références mais aussi du manque de documentation proposant des exemples destinés au débutant.

Ce document est conçu pour traiter, en détail mais de façon compréhensible, toute une panoplie de structures de données que vous pourriez être amené à développer. Il devrait aussi servir de livre de recettes donnant des exemples. De cette façon, lorsque vous avez besoin de créer l'une de ces structures de données complexes, vous pouvez simplement piquer, chaparder ou dérober un exemple de ce guide.

Jetons un oeil en détail à chacune de ces constructions possibles. Il existe des sections séparées pour chacun des cas suivants :

- tableaux de tableaux
- hachages de tableaux
- tableaux de hachages
- hachages de hachages
- constructions plus élaborées

Mais pour le moment, intéressons-nous aux problèmes communs à tous ces types de structures de données.

5.2 RÉFÉRENCES

La chose la plus importante à comprendre au sujet de toutes les structures de données en Perl - y compris les tableaux multidimensionnels - est que même s'ils peuvent paraître différents, les @TABLEAUX et les %HACHAGES en Perl sont tous unidimensionnels en interne. Ils ne peuvent contenir que des valeurs scalaires (c'est-à-dire une chaîne, un nombre ou une référence). Ils ne peuvent pas contenir directement d'autres tableaux ou hachages, mais ils contiennent à la place des *références* à des tableaux ou des hachages.

Vous ne pouvez pas utiliser une référence à un tableau ou à un hachage exactement de la même manière que vous le feriez d'un vrai tableau ou d'un vrai hachage. Pour les programmeurs C ou C++, peu habitués à distinguer les tableaux et les pointeurs vers des tableaux, ceci peut être déconcertant. Si c'est le cas, considérez simplement la différence entre une structure et un pointeur vers une structure.

Vous pouvez (et devriez) en lire plus au sujet des références dans *perlref*. Brièvement, les références ressemblent assez à des pointeurs sachant vers quoi ils pointent (les objets sont aussi une sorte de référence, mais nous n'en aurons pas besoin pour le moment). Ceci signifie que lorsque vous avez quelque chose qui vous semble être un accès à un tableau ou à un hachage à deux dimensions ou plus, ce qui se passe réellement est que le type de base est seulement une entité unidimensionnelle qui contient des références vers le niveau suivant. Vous pouvez juste l'*utiliser* comme si c'était une entité à deux dimensions. C'est aussi en fait la façon dont fonctionnent presque tous les tableaux multidimensionnels en C.

```
$array[7][12]           # tableau de tableaux
$array[7]{string}      # tableau de hachages
$hash{string}[7]      # hachage de tableaux
$hash{string}{'another string'} # hachage de hachages
```

Maintenant, puisque le niveau supérieur ne contient que des références, si vous essayez d'imprimer votre tableau avec une simple fonction `print()`, vous obtiendrez quelque chose qui n'est pas très joli, comme ceci :

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

C'est parce que Perl ne déréférence (presque) jamais implicitement vos variables. Si vous voulez atteindre la chose à laquelle se réfère une référence, alors vous devez le faire vous-même soit en utilisant des préfixes d'indication de type, comme `$$blah`, `@$blah`, `@$blah[$i]`, soit des flèches de pointage postfixes, comme `$a->[3]`, `$h->{fred}`, ou même `$ob->method()->[3]`.

5.3 ERREURS COURANTES

Les deux erreurs les plus communes faites lors de l'affectation d'une donnée de type tableau de tableaux est soit l'affectation du nombre d'éléments du tableau (NdT : au lieu du tableau lui-même bien sûr), soit l'affectation répétée d'une même référence mémoire. Voici le cas où vous obtenez juste le nombre d'éléments au lieu d'un tableau imbriqué :

```
for $i (1..10) {
    @list = somefunc($i);
    $AoA[$i] = @list;      # FAUX !
}
```

C'est le cas simple où l'on affecte un tableau à un scalaire et où l'on obtient le nombre de ses éléments. Si c'est vraiment bien ce que vous désirez, alors vous pourriez être un poil plus explicite à ce sujet, comme ceci :

```
for $i (1..10) {
    @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Voici le cas où vous vous référez encore et encore au même emplacement mémoire :

```

for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;    # FAUX !
}

```

Allons bon, quel est le gros problème ? Cela semble bon, n'est-ce pas ? Après tout, je viens de vous dire que vous aviez besoin d'un tableau de références, alors flûte, vous m'en avez créé un !

Malheureusement, bien que cela soit vrai, cela ne marche pas. Toutes les références dans @AoA se réfère au *même endroit*, et elles contiendront toutes par conséquent ce qui se trouvait en dernier dans @array ! Le problème est similaire à celui du programme C suivant :

```

#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
          dp->pw_name, rp->pw_name);
}

```

Ce qui affichera :

```

daemon name is daemon
root name is daemon

```

Le problème est que rp et dp sont des pointeurs vers le même emplacement en mémoire ! En C, vous devez utiliser malloc() pour vous réserver de la mémoire. En Perl, vous devez utiliser à la place le constructeur de tableau [] ou le constructeur de hachages {}. Voici la bonne façon de procéder :

```

for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}

```

Les crochets créent une référence à un nouveau tableau avec une *copie* de ce qui se trouve dans @array au moment de l'affectation. C'est ce que vous désiriez.

Notez que ceci produira quelque chose de similaire, mais c'est bien plus dur à lire :

```

for $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}

```

Est-ce la même chose ? Eh bien, peut-être que oui, peut-être que non. La différence subtile est que lorsque vous affectez quelque chose entre crochets, vous êtes sûr que ce sera toujours une nouvelle référence contenant une nouvelle *copie* des données. Ça ne se passera pas forcément comme ça avec le déréférencement de @{\$AoA[\$i]} du côté gauche de l'affectation. Tout dépend si \$AoA[\$i] était pour commencer indéfini, ou s'il contenait déjà une référence. Si vous aviez déjà peuplé @AoA avec des références, comme dans

```

$AoA[3] = \@another_array;

```

alors l'affectation avec l'indirection du côté gauche utiliserait la référence existante déjà présente :

```

 @{$AoA[3]} = @array;

```

Bien sûr, ceci *aurait* l'effet "intéressant" de démolir @another_list (Avez-vous déjà remarqué que lorsqu'un programmeur dit que quelque chose est "intéressant", au lieu de dire "intrigant", alors ça signifie que c'est "ennuyeux", "difficile", ou les deux ? :-).

Donc, souvenez-vous juste de toujours utiliser le constructeur de tableau ou de hachage [] ou {} et tout ira bien pour vous, même si ce n'est pas la solution optimale.

De façon surprenante, la construction suivante qui a l'air dangereuse marchera en fait très bien :


```
for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

C'est parce que `my()` est plus une expression utilisée lors de l'exécution qu'une déclaration de compilation *en elle-même*. Cela signifie que la variable `my()` est recrée de zéro chaque fois que l'on traverse la boucle. Donc même si on a l'impression que vous stockez la même référence de variable à chaque fois, ce n'est pas le cas ! C'est une distinction subtile qui peut produire un code plus efficace au risque de tromper tous les programmeurs sauf les plus expérimentés. Je déconseille donc habituellement de l'enseigner aux débutants. En fait, sauf pour passer des arguments à des fonctions, j'aime rarement voir l'opérateur "donne-moi-une-référence" (backslash) utilisé dans du code. À la place, je conseille aux débutants (et à la plupart d'entre nous) d'essayer d'utiliser les constructeurs bien plus compréhensibles `[]` et `{}` au lieu de se reposer sur une astuce lexicale (ou dynamique) et un comptage de référence caché pour faire ce qu'il faut en coulisses. En résumé :

```
$AoA[$i] = [ @array ];      # habituellement mieux
$AoA[$i] = \@array;        # perilleux ; tableau declare
                             # avec my() ? Comment ?
@{ $AoA[$i] } = @array;    # bien trop astucieux pour la plupart
                             # des programmeurs
```

5.4 AVERTISSEMENT SUR LA PRÉCÉDENCE

En parlant de choses comme `@{$AoA[$i]}`, les expressions suivantes sont en fait équivalentes :

```
$aref->[2][2]      # clair
$$aref[2][2]      # troublant
```

C'est dû aux règles de précédence de Perl qui rend les cinq préfixes déréférenciers (qui ont l'air de jurons : `$ @ * % &`) plus prioritaires que les accolades et les crochets d'indilage postfixes ! Ceci sera sans doute un grand choc pour le programmeur C ou C++, qui est habitué à utiliser `*a[i]` pour indiquer ce qui est pointé par le *i-ème* élément de `a`. C'est-à-dire qu'ils prennent d'abord l'indice, et ensuite seulement déréférencent la structure à cet indice. C'est bien en C, mais nous ne parlons pas de C.

La construction apparemment équivalente en Perl, `$$aref[$i]` commence par déréférencer `$aref`, le faisant prendre `$aref` comme une référence à un tableau, et puis déréférence cela, et finalement vous donne la valeur du *i-ème* élément du tableau pointé par `$AoA`. Si vous vouliez la notion équivalente en C, vous devriez écrire `$$AoA[$i]` pour forcer l'évaluation de `$AoA[$i]` avant le premier déréférencier `$`.

5.5 POURQUOI TOUJOURS UTILISER `use strict`

Ce n'est pas aussi effrayant que ça en a l'air, détendez-vous. Perl possède un certain nombre de caractéristiques qui vont vous aider à éviter les pièges les plus communs. La meilleure façon de ne pas être troublé est de commencer les programmes ainsi :

```
#!/usr/bin/perl -w
use strict;
```

De cette façon, vous serez forcé de déclarer toutes vos variables avec `my()` et aussi d'interdire tout "déréférencement symbolique" accidentel. Par conséquent, si vous faites ceci :

```
my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

Le compilateur marquera immédiatement ceci comme une erreur *lors de la compilation*, car vous accédez accidentellement à `@aref` qui est une variable non déclarée, et il vous proposerait d'écrire à la place :

```
print $aref->[2][2]
```

5.6 DÉBOGAGE

Avant la version 5.002, le débogueur standard de Perl ne faisait pas un très bon travail lorsqu'il devait imprimer des structures de données complexes. Avec la version 5.002 et les suivantes, le débogueur inclut plusieurs nouvelles caractéristiques, dont une édition de la ligne de commande ainsi que la commande `x` pour afficher les structures de données complexes. Par exemple, voici la sortie du débogueur avec l'affectation à `$AoA` ci-dessus :

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
  0  ARRAY(0x1f0a24)
     0  'fred'
     1  'barney'
     2  'pebbles'
     3  'bambam'
     4  'dino'
  1  ARRAY(0x13b558)
     0  'homer'
     1  'bart'
     2  'marge'
     3  'maggie'
  2  ARRAY(0x13b540)
     0  'george'
     1  'jane'
     2  'elroy'
     3  'judy'
```

5.7 EXEMPLES DE CODE

Présentés avec peu de commentaires (ils auront leurs propres pages de manuel un jour), voici de courts exemples de code illustrant l'accès à divers types de structures de données.

5.8 TABLEAUX DE TABLEAUX

5.8.1 Déclaration d'un TABLEAU DE TABLEAUX

```
@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);
```

5.8.2 Génération d'un TABLEAU DE TABLEAUX

```
# lecture dans un fichier
while ( <> ) {
    push @AoA, [ split ];
}

# appel d'une fonction
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# utilisation de variables temporaires
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# ajout dans une rangée existante
push @{$AoA[0]}, "wilma", "betty";
```

5.8.3 Accès et affichage d'un TABLEAU DE TABLEAUX

```
# un element
$AoA[0][0] = "Fred";

# un autre element
$AoA[1][1] =~ s/(\w)/\u$1/;

# affiche le tout avec des refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# affiche le tout avec des indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# affiche tous les elements un par un
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

5.9 HACHAGE DE TABLEAUX

5.9.1 Déclaration d'un HACHAGE DE TABLEAUX

```
%HoA = (
    flintstones    => [ "fred", "barney" ],
    jetsons        => [ "george", "jane", "elroy" ],
    simpsons       => [ "homer", "marge", "bart" ],
);
```

5.9.2 Génération d'un HACHAGE DE TABLEAUX

```
# lecture dans un fichier
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# lecture dans un fichier avec plus de variables temporaires
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# appel d'une fonction qui retourne une liste
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}
```

```
# idem, mais en utilisant des variables temporaires
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}

# ajout de nouveaux membres a une famille existante
push @{$HoA{"flintstones"}}, "wilma", "betty";
```

5.9.3 Accès et affichage d'un HACHAGE DE TABLEAUX

```
# un element
$HoA{flintstones}[0] = "Fred";

# un autre element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# affichage du tout
foreach $family ( keys %HoA ) {
    print "$family: @{$HoA{$family}}\n"
}

# affichage du tout avec des indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. ${HoA{$family}} ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# affichage du tout trie par le nombre de membres
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{$HoA{$family}}\n"
}

# affichage du tout trie par le nombre de membres et le nom
foreach $family ( sort {
    @{$HoA{$b}} <=> @{$HoA{$a}}
    ||
    $a cmp $b
} keys %HoA )
{
    print "$family: ", join(", ", sort @{$HoA{$family}} ), "\n";
}
```

5.10 TABLEAUX DE HACHAGES

5.10.1 Déclaration d'un TABLEAU DE HACHAGES

```
@AoH = (
    {
        Lead    => "fred",
        Friend  => "barney",
    },
    {
        Lead    => "george",
        Wife    => "jane",
        Son     => "elroy",
    }
)
```

```

    },
    {
        Lead    => "homer",
        Wife    => "marge",
        Son     => "bart",
    }
);

```

5.10.2 Génération d'un TABLEAU DE HACHAGES

```

# lecture dans un fichier
# format : LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

# lecture dans un fichier sans variable temporaire
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    push @AoH, { split /\s+=/ };
}

# appel d'une fonction qui retourne une liste clé/valeur, comme
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# idem, mais sans variables temporaires
while ( <> ) {
    push @AoH, { parsepairs($_) };
}

# ajout d'un couple clé/valeur à un element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";

```

5.10.3 Accès et affichage d'un TABLEAU DE HACHAGES

```

# un element
$AoH[0]{lead} = "fred";

# un autre element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# affichage du tout avec des refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

```

```

# affichage du tout avec des indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

# affichage du tout element par element
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}

```

5.11 HACHAGES DE HACHAGES

5.11.1 Déclaration d'un HACHAGE DE HACHAGES

```

%HoH = (
    flintstones => {
        lead    => "fred",
        pal     => "barney",
    },
    jetsons    => {
        lead    => "george",
        wife    => "jane",
        "his boy" => "elroy",
    },
    simpsons   => {
        lead    => "homer",
        wife    => "marge",
        kid     => "bart",
    },
);

```

5.11.2 Génération d'un HACHAGE DE HACHAGES

```

# lecture dans un fichier
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

# lecture dans un fichier, encore plus de variables temporaires
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

```

```

# appel d'une fonction qui retourne un hachage clé/valeur
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# idem, mais en utilisant des variables temporaires
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# ajout de nouveaux membres a une famille existante
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}

```

5.11.3 Accès et affichage d'un HACHAGE DE HACHAGES

```

# un element
$HoH{flintstones}{wife} = "wilma";

# un autre element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# affichage du tout
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# affichage du tout un peu trie
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# affichage du tout trie par le nombre de membres
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# etablissement d'un ordre de tri (rang) pour chaque role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

```

```
# maintenant affiche le tout trie par le nombre de membres
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # et affichage selon l'ordre de tri (rang)
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
}
```

5.12 ENREGISTREMENTS PLUS ÉLABORÉS

5.12.1 Déclaration d'ENREGISTREMENTS PLUS ÉLABORÉS

Voici un exemple montrant comment créer et utiliser un enregistrement dont les champs sont de types différents :

```
$rec = {
    TEXT      => $string,
    SEQUENCE => [ @old_values ],
    LOOKUP   => { %some_table },
    THATCODE => \&some_function,
    THISCODE => sub { $_[0] ** $_[1] },
    HANDLE   => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# attention aux accolades de bloc supplémentaires sur la ref fh
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

5.12.2 Déclaration d'un HACHAGE D'ENREGISTREMENTS COMPLEXES

```
%TV = (
    flintstones => {
        series    => "flintstones",
        nights    => [ qw(monday thursday friday) ],
        members   => [
            { name => "fred",    role => "lead", age => 36, },
            { name => "wilma",   role => "wife", age => 31, },
            { name => "pebbles", role => "kid",  age => 4,  },
        ],
    },
),
```



```

jetsons    => {
  series   => "jetsons",
  nights   => [ qw(wednesday saturday) ],
  members  => [
    { name => "george",  role => "lead", age => 41, },
    { name => "jane",    role => "wife", age => 39, },
    { name => "elroy",   role => "kid",  age =>  9, },
  ],
},

simpsons   => {
  series   => "simpsons",
  nights   => [ qw(monday) ],
  members  => [
    { name => "homer",  role => "lead", age => 34, },
    { name => "marge",  role => "wife", age => 37, },
    { name => "bart",   role => "kid",  age => 11, },
  ],
},
);

```

5.12.3 Génération d'un HACHAGE D'ENREGISTREMENTS COMPLEXES

```

# lecture d'un fichier
# c'est plus facile a faire quand on dispose du fichier lui-meme
# dans le format de donnees brut explicite ci-dessus. perl analyse
# joyeusement les structures de donnees complexes si elles sont
# declarees comme des donnees, il est donc parfois plus facile de
# faire ceci

# voici un assemblage morceau par morceau
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# on presume que ce fichier a la syntaxe champ=valeur
while (<>) {
  %fields = split /\s=/+;
  push @members, { %fields };
}
$rec->{members} = [ @members ];

# maintenant on se rappelle du tout
$TV{ $rec->{series} } = $rec;

#####
# maintenant, vous pourriez vouloir creer des champs
# supplementaires interessants incluant des pointeurs vers
# l'interieur de cette meme structure de donnees pour que
# si l'on en change un morceau, il se retrouve change partout
# ailleurs, comme par exemple un champ {kids} (enfants, NDT)
# qui serait une reference vers un tableau des enregistrements
# des enfants, sans avoir d'enregistrements dupliques et donc
# des soucis de mise a jour.
#####
foreach $family (keys %TV) {
  $rec = $TV{$family}; # pointeur temporaire
  @kids = ();
  for $person ( @{ $rec->{members} } ) {

```

```

        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # SOUVENEZ-VOUS : $rec et $TV{$family} pointent sur les memes donnees !!
    $rec->{kids} = [ @kids ];
}

# vous avez copie le tableau, mais le tableau lui-meme contient des
# pointeurs vers des objets qui n'ont pas été copiés. Cela veut
# dire que si vous vieillissez bart en faisant

$TV{simpsons}{kids}[0]{age}++;

# alors ça changerait aussi
print $TV{simpsons}{members}[2]{age};

# car $TV{simpsons}{kids}[0] et $TV{simpsons}{members}[2]
# pointent tous deux vers la meme table de hachage anonyme sous-jacente

# imprime le tout
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{ $TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n";
}

```

5.13 Liens avec les bases de données

Il n'est pas possible de lier facilement une structure de données multiniveaux (tel qu'un hachage de hachages) à un fichier de base de données (fichier dbm). Le premier problème est que tous ces fichiers, à l'exception des GBM et des DB de Berkeley, ont des limitations de taille, mais il y a d'autres problèmes dus la manière dont les références sont représentées sur le disque. Il existe un module expérimental, le module MLDBM, qui essaye de combler partiellement ce besoin. Pour obtenir le code source du module MLDBM allez voir sur le site CPAN le plus proche comme décrit dans *perlmodlib*.

5.14 VOIR AUSSI

perlref, *perllol*, *perldata* et *perlobj*.

5.15 AUTEUR

Tom Christiansen <tchrist@perl.com>

Dernière mise à jour : Wed Oct 23 04:57:50 MET DST 1996

5.16 TRADUCTION

5.16.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

5.16.2 Traducteur

Roland Trique <*roland.trique@free.fr*>

5.16.3 Relecture

Pascal Ethvignot <*pascal@encelade.frmug.org*>, Etienne Gauthier <*egauthie@capgemini.fr*>, Paul Gaborit (Paul.Gaborit @ enstimac.fr).

Chapitre 6

perl lol

Manipulation des tableaux de tableaux en Perl

6.1 DESCRIPTION

6.1.1 Déclaration et accès aux tableaux de tableaux

La chose la plus simple à construire est un tableau de tableaux (parfois appelé de façon peu précise une liste de listes). C'est raisonnablement facile à comprendre et presque tout ce qui s'y applique pourra aussi être appliqué par la suite aux structures de données plus fantaisistes.

Un tableau de tableau est juste un bon vieux tableau `@TdT` auquel vous pouvez accéder avec deux indices, comme `$TdT[3][2]`. Voici une déclaration du tableau :

```
# affecte à notre tableau un tableau de références à des tableaux
@TdT = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);

print $TdT[2][2];
bart
```

Maintenant, vous devez faire bien attention au fait que les parenthèses extérieures sont bien des parenthèses et pas des accolades ou des crochets. C'est parce que vous affectez dans un `@tableau`. Vous avez donc besoin de parenthèses. Si vous *n'aviez pas* voulu que cela soit un `@TdT`, mais plutôt une référence à lui, vous auriez pu faire quelque chose du style de ceci :

```
# affecte une référence à une liste de références de liste
$ref_to_TdT = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_TdT->[2][2];
```

Notez que le type des parenthèses extérieures a changé et donc notre syntaxe d'accès aussi. C'est parce que, contrairement au C, en Perl vous ne pouvez pas librement échanger les tableaux et leurs références. `$ref_to_TdT` est une référence à un tableau, tandis que `@TdT` est un tableau proprement dit. De la même manière, `$TdT[2]` n'est pas un tableau, mais une référence à un tableau. Ainsi donc vous pouvez écrire ceci :

```
$TdT[2][2]
$ref_to_TdT->[2][2]
```

au lieu de devoir écrire ceci :

```
$TdT[2]->[2]
$ref_to_TdT->[2]->[2]
```

Vous pouvez le faire car la règle dit que, entre des crochets ou des accolades adjacents, vous êtes libre d'omettre la flèche de déréférencement. Mais vous ne pouvez pas faire cela pour la toute première flèche si c'est un scalaire contenant une référence, ce qui signifie que \$ref_to_TdT en a toujours besoin.

6.1.2 Développer la vôtre

Tout ceci est bel et bien pour la déclaration d'une structure de données fixe, mais si vous voulez ajouter de nouveaux éléments à la volée, ou tout construire à partir de zéro ?

Tout d'abord, étudions sa lecture à partir d'un fichier. C'est quelque chose comme ajouter une rangée à la fois. Nous présumerons qu'il existe un fichier tout simple dans lequel chaque ligne est une rangée et chaque mot un élément. Voici la bonne façon de le faire si vous essayez de développer un tableau @TdT les contenant tous :

```
while (<>) {
    @tmp = split;
    push @TdT, [ @tmp ];
}
```

Vous auriez aussi pu charger tout cela dans une fonction :

```
for $i ( 1 .. 10 ) {
    $TdT[$i] = [ somefunc($i) ];
}
```

Ou vous auriez pu avoir une variable temporaire traînant dans le coin et contenant le tableau.

```
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $TdT[$i] = [ @tmp ];
}
```

Il est très important que vous vous assuriez d'utiliser le constructeur de référence de tableau []. C'est parce que ceci serait très mauvais :

```
$TdT[$i] = @tmp;
```

Voyez vous, affecter comme ceci un tableau nommé à un scalaire ne fait que compter le nombre d'éléments dans @tmp, ce qui n'est probablement pas ce que vous désirez.

Si vous utilisez `use strict`, vous devrez ajouter quelques déclarations pour que tout fonctionne :

```
use strict;
my(@TdT, @tmp);
while (<>) {
    @tmp = split;
    push @TdT, [ @tmp ];
}
```

Bien sûr, vous n'avez pas du tout besoin de donner un nom au tableau temporaire :

```
while (<>) {
    push @TdT, [ split ];
}
```

Vous n'êtes pas non plus obligé d'utiliser `push()`. Vous pourriez juste faire une affectation directe si vous savez où vous voulez le mettre :

```

my (@TdT, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $TdT[$i] = [ split ' ', $line ];
}

```

ou même juste :

```

my (@TdT, $i);
for $i ( 0 .. 10 ) {
    $TdT[$i] = [ split ' ', <> ];
}

```

Vous devriez en général lorgner d'un regard mauvais l'usage de fonctions qui peuvent potentiellement retourner des listes dans un contexte scalaire sans le formuler explicitement. Ceci sera plus clair pour le lecteur de passage :

```

my (@TdT, $i);
for $i ( 0 .. 10 ) {
    $TdT[$i] = [ split ' ', scalar(<>) ];
}

```

Si vous voulez utiliser une variable \$ref_to_TdT comme référence à un tableau, vous devez faire quelque chose comme ceci :

```

while (<>) {
    push @$ref_to_TdT, [ split ];
}

```

Maintenant vous pouvez ajouter de nouvelles rangées. Et pour ajouter de nouvelles colonnes ? Si vous traitez juste des matrices, le plus facile est souvent d'utiliser une simple affectation :

```

for $x (1 .. 10) {
    for $y (1 .. 10) {
        $TdT[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $TdT[$x][20] += func2($x);
}

```

Peu importe que ces éléments soient déjà là ou pas : ils seront créés joyeusement pour vous et, si besoin, les éléments intermédiaires seront initialisés à undef.

Si vous voulez juste en ajouter à une rangée, vous devrez faire quelque chose ayant l'air un peu plus bizarre :

```

# ajoute de nouvelles colonnes à une rangée existante
push @{$TdT[0]}, "wilma", "betty";

```

Remarquez qu'on *ne pourrait pas* juste dire :

```

push $TdT[0], "wilma", "betty"; # FAUX !

```

En fait, cela ne se compilerait même pas. Pourquoi donc ? Parce que l'argument de push() doit être un véritable tableau, et non pas une simple référence.

6.1.3 Accès et sortie

Maintenant il est temps de sortir votre structure de données. Comment allez-vous faire une telle chose ? Eh bien, si vous voulez uniquement l'un des éléments, c'est trivial :

```
print $TdT[0][0];
```

Si vous voulez sortir toute la chose, toutefois, vous ne pouvez pas dire :

```
print @TdT;          # FAUX
```

car vous obtiendrez juste la liste des références et perl ne déréférencera jamais automatiquement les choses pour vous. Au lieu de cela, vous devez faire tourner une boucle ou deux. Ceci imprime toute la structure, en utilisant la construction for() dans le style du shell pour boucler d'un bout à l'autre de l'ensemble des indices extérieurs.

```
for $aref ( @TdT ) {
    print "\t [ @$aref ],\n";
}
```

Si vous voulez garder la trace des indices, vous pouvez faire ceci :

```
for $i ( 0 .. $#TdT ) {
    print "\t elt $i is [ @{$TdT[$i]} ],\n";
}
```

ou peut-être même ceci. Remarquez la boucle intérieure.

```
for $i ( 0 .. $#TdT ) {
    for $j ( 0 .. ${#TdT[$i]} ) {
        print "elt $i $j is $TdT[$i][$j]\n";
    }
}
```

Comme vous pouvez le voir, cela devient un peu compliqué. C'est pourquoi il est parfois plus facile de prendre une variable temporaire en chemin :

```
for $i ( 0 .. $#TdT ) {
    $aref = $TdT[$i];
    for $j ( 0 .. ${#$aref} ) {
        print "elt $i $j is $TdT[$i][$j]\n";
    }
}
```

Hmm... c'est encore un peu laid. Pourquoi pas ceci :

```
for $i ( 0 .. $#TdT ) {
    $aref = $TdT[$i];
    $n = @$aref - 1;
    for $j ( 0 .. $n ) {
        print "elt $i $j is $TdT[$i][$j]\n";
    }
}
```

6.1.4 Tranches

Si vous voulez accéder à une tranche (une partie d'une rangée) d'un tableau multidimensionnel, vous allez devoir faire un peu d'indigage fantaisiste. Car, tandis que nous avons un joli synonyme pour les éléments seuls via la flèche de pointeur pour le déréférencement, il n'existe pas de telle commodité pour les tranches (souvenez-vous, bien sûr, que vous pouvez toujours écrire une boucle pour effectuer une opération sur une tranche).

Voici comment faire une opération en utilisant une boucle. Nous supposons avoir une variable @TdT comme précédemment.

```
@part = ();
$x = 4;
for ($y = 7; $y < 13; $y++) {
    push @part, $TdT[$x][$y];
}
```

Cette même boucle peut être remplacée par une opération de tranche :

```
@part = @{ $TdT[4] } [ 7..12 ];
```

mais comme vous pouvez l'imaginer, c'est plutôt rude pour le lecteur.

Et si vous vouliez une *tranche à deux dimensions*, telle que \$x varie dans 4..8 et \$y dans 7 à 12 ? Hmm... voici la façon simple :

```
@newTdT = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newTdT[$x - $startx][$y - $starty] = $TdT[$x][$y];
    }
}
```

Nous pouvons réduire une partie du bouclage via des tranches.

```
for ($x = 4; $x <= 8; $x++) {
    push @newTdT, [ @{ $LoL[$x] } [ 7..12 ] ];
}
```

Si vous faisiez des transformations schwartziennes, vous auriez probablement choisi map pour cela :

```
@newTdT = map { [ @{ $TdT[$_] } [ 7..12 ] ] } 4 .. 8;
```

Bien sûr, si votre directeur vous accuse de rechercher la sécurité de l'emploi (ou l'insécurité rapide) à l'aide d'un code indéchiffrable, il sera difficile d'argumenter. :-) Si j'étais vous, je mettrais cela dans une fonction :

```
@newTdT = splice_2D( \@LoL, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;      # réf à un tableau de réfs. de tableau !
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;

    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

6.2 VOIR AUSSI

perldata, *perlref*, *perldsc*.

6.3 AUTEUR

Tom Christiansen <*tchrist@perl.com*>

Dernière mise à jour : Thu Jun 4 16:16:23 MDT 1998

6.4 TRADUCTION

6.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

6.4.2 Traducteur

Roland Trique <*roland.trique@free.fr*>.

6.4.3 Relecture

Régis Julié <*regis.julie@cetelem.fr*>. Paul Gaborit (Paul.Gaborit at enstimac.fr).

Chapitre 7

perlrequick

Les expressions rationnelles Perl pour les impatientes

7.1 DESCRIPTION

Ce document décrit les bases nécessaires à la compréhension, la création et l'utilisation des expressions rationnelles ou régulières (abrégé en regex) en Perl.

7.2 Le guide

7.2.1 Reconnaissance de mot simple

L'expression rationnelle la plus simple est juste un mot ou, plus généralement, une chaîne de caractères. Une regex constituée d'un mot est reconnue dans (ou correspond avec) toutes les chaînes qui contiennent ce mot :

```
"Salut tout le monde" =~ /monde/; # correspondance
```

Dans cette instruction, `monde` est la regex et les `//` qui l'entourent demandent à perl de rechercher une correspondance dans la chaîne. L'opérateur `=~` applique la recherche à la chaîne placée à gauche et produit la valeur vraie si la regex correspond ou la valeur fausse sinon. Dans notre cas, `monde` correspond au quatrième mot de "Salut tout le monde". Donc l'expression est vraie. Ce concept a plusieurs usages.

Des expressions de ce type sont utiles dans des conditions :

```
print "Correspondance\n" if "Salut tout le monde" =~ /monde/;
```

Le sens de l'expression peut être inversé en utilisant l'opérateur `!~` :

```
print "Pas de correspondance\n" if "Salut tout le monde" !~ /monde/;
```

La chaîne littérale dans la regex peut être remplacée par une variable :

```
$mot = "monde";  
print "Correspondance\n" if "Salut tout le monde" =~ /$mot/;
```

Si vous voulez chercher dans `$_`, la partie `$_ =~` peut être omise :

```
$_ = "Salut tout le monde";  
print "Correspondance\n" if /monde/;
```

Finalement, les délimiteurs par défaut `//` pour une recherche de correspondance peuvent être remplacés par des délimiteurs arbitraires en les préfixant par un `'m'` :

```
"Salut le monde" =~ m!monde!; # correspondance, délimité par '!'
"Salut le monde" =~ m{monde}; # correspondance, remarquez la paire '{}'
"/usr/bin/perl" =~ m"/perl"; # correspondance après '/usr/bin',
                               # '/' devient un caractère ordinaire
```

Les regex doivent correspondre *exactement* à une partie de la chaîne pour être reconnue :

```
"Salut le monde" =~ /Monde/; # pas de correspondance, casse différente
"Salut le monde" =~ /e m/;   # correspondance, ' ' est un caractère ordinaire
"Salut le monde" =~ /monde /; # pas de correspondance, pas de ' ' à la fin
```

La reconnaissance a lieu le plus tôt possible dans la chaîne :

```
"Salut le monde" =~ /l/;    # reconnaît le 'l' dans 'Salut'
"La peste est là" =~ /est/; # reconnaît le 'est' dans 'peste'
```

Certains caractères ne peuvent être utilisés tels quels dans une regex. Ces caractères, appelés **meta-caractères**, sont réservés pour des notations spéciales dans les expressions rationnelles. Les voici :

```
{ } [ ] ( ) ^ $ . | * + ? \
```

Un meta-caractère peut-être utilisé en le préfixant par un backslash (une barre oblique inversée) :

```
"2+2=4" =~ /2+2/;    # pas de correspondance, + est un meta-caractère
"2+2=4" =~ /2\+2/;   # correspondance, \+ est traité comme un + ordinaire
'C:\WIN32' =~ /C:\\WIN/; # correspondance
"/usr/bin/perl" =~ /\usr\bin\perl/; # correspondance
```

Dans ce dernier exemple, le symbole divisé / est aussi préfixé par un backslash car il est utilisé comme délimiteur par l'expression rationnelle.

Les caractères ASCII non affichables sont représentés par des **séquences d'échappement**. Les exemples courants sont `\t` pour une tabulation, `\n` pour un passage à la ligne et `\r` pour un retour chariot. Les octets quelconques sont représentés par une séquence d'échappement en octal (comme `\033`) ou en hexadécimal (comme `\x1B`) :

```
"1000\t2000" =~ m(0\t2) # correspondance
"chat"       =~ /\143\150\x61\x74/ # correspondance,
                                     # mais 'chat' est écrit bizarrement
```

Les expressions rationnelles sont traitées quasiment comme des chaînes entre guillemets. Donc l'interpolation des variables fonctionne :

```
$foo = 'son';
'caisson' =~ /cais$foo/; # correspondance
'sonnet'  =~ /${foo}net/; # correspondance
```

Dans toutes les expressions rationnelles qui précèdent, si la regex est reconnue quelque part dans la chaîne, on considère qu'il y a correspondance. Pour spécifier *où* doit avoir lieu la reconnaissance, vous pouvez utiliser les meta-caractères d'**ancrage** `^` et `$`. L'ancrage `^` est reconnu au début de la chaîne alors que l'ancrage `$` est reconnu à la fin de la chaîne ou juste avant une fin de ligne à la fin de la chaîne. Quelques exemples :

```
"housekeeper" =~ /keeper/;    # correspondance
"housekeeper" =~ /^keeper/;   # pas de correspondance
"housekeeper" =~ /keeper$/;   # correspondance
"housekeeper\n" =~ /keeper$/; # correspondance
"housekeeper" =~ /^housekeeper$/; # correspondance
```

7.2.2 Utilisation des classes de caractères

Une **classe de caractères** définit un ensemble de caractères acceptables en un point particulier de l'expression rationnelle. Une classe de caractères s'exprime par une paire de crochets [...] contenant l'ensemble des caractères acceptables. Voici quelques exemples :

```
/rame/;          # reconnaît 'rame'
/[clr]ame/;     # reconnaît 'came', 'lame' ou 'rame'
"abc" =~ /[cab]/; # reconnaît 'a'
```

Dans la dernière instruction, bien que 'c' soit le premier caractère de la classe, le premier endroit où cette expression rationnelle peut être reconnue est le 'a'.

```
/[oO][uU][iI]/; # reconnaît 'oui' indépendamment de la casse
                # 'oui', 'Oui', 'OUI', etc.
/oui/i;         # reconnaît aussi 'oui' indépendamment de la casse
```

Le dernier exemple démontre l'usage du modificateur 'i' qui permet une mise en correspondance indépendante de la casse (majuscule/minuscule).

Les classes de caractères ont elles aussi leurs caractères normaux et spéciaux mais ce ne sont pas les mêmes qu'à l'extérieur d'une classe. Les caractères spéciaux sont -]\^\$. Pour les rendre normaux, il faut les préfixer par \ :

```
/[\]c]def/; # correspond à 'def' ou 'cdef'
$x = 'clr';
/>$x]ame/; # correspond à 'came', 'lame' ou 'rame'
/>\$x]ame/; # correspond à '$ame' or 'xame'
/>\$x]ame/; # correspond à '\ame', 'came', 'lame' ou 'rame'
```

Le caractère spécial '-' agit à l'intérieur d'une classe comme un opérateur d'intervalle. Donc les classes peu maniables [0123456789] et [abcde...xyz] deviennent [0-9] et [a-z] :

```
/item[0-9]/; # reconnaît 'item0' ou 'item1' ... ou 'item9'
/[0-9a-fA-F]/; # reconnaît un chiffre hexadécimal
```

Si '-' est le premier ou le dernier caractère d'une classe de caractères, il est traité comme un caractère ordinaire.

Le caractère ^ est spécial en première position de la classe. Il indique alors une **classe de caractères complémentaire** qui reconnaît tous les caractères sauf ceux présents entre les crochets. Qu'elle soit de la forme [...] ou [^...], une classe de caractères doit correspondre à un caractère sinon la reconnaissance échoue. Donc :

```
/[^a]at/; # ne reconnaît ni 'aat' ni 'at', mais reconnaît
          # 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # reconnaît un caractère non numérique
/[a^]at/; # reconnaît 'aat' ou '^at'; dans ce cas '^' est ordinaire
```

Perl propose plusieurs abréviations pour des classes de caractères courantes :

- \d est un chiffre et est équivalent à [0-9]
- \s est un blanc et est équivalent à [\ \t\r\n\f]
- \w est caractère *mot* (alphanumérique ou _) et est équivalent à [0-9a-zA-Z_]
- \D est la négation de \d; il représente tout autre caractère qu'un chiffre [^0-9]
- \S est la négation de \s [^\s]
- \W est la négation de \w [^\w]
- Le point '.' reconnaît n'importe quel caractère sauf "\n".

Les abréviations `\d\s\w\D\S\W` peuvent être utilisées à l'extérieur ou à l'intérieur d'une classe de caractères. Quelques exemples :

```
\d\d:\d\d:\d\d/; # reconnaît une heure au format hh:mm:ss
/[\d\s]/;        # reconnaît un chiffre ou un blanc
/\w\W\w/;        # reconnaît un caractère mot suivi d'un caractère
                  # non mot, suivi d'un caractère mot
/..rt/;          # reconnaît deux caractères quelconques suivis de 'rt'
/fin\./;         # reconnaît 'fin.'
/fin[.]/;        # idem, reconnaît 'fin.'
```

L'ancre `\b` est reconnue à la limite de mot : entre un caractère mot et un caractère non mot (entre `\w\W` ou entre `\W\w`).

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/; # reconnaît cat dans 'catenates'
$x =~ /cat\b/; # reconnaît cat dans 'housecat'
$x =~ /\bcat\b/; # reconnaît 'cat' en fin de chaîne
```

Dans le dernier exemple, la fin de la chaîne est considérée comme une limite de mot.

7.2.3 Reconnaître ceci ou cela

Nous pouvons reconnaître différentes chaînes grâce au meta-caractère d'alternative `|`. Pour reconnaître `chien` ou `chat`, nous pouvons utiliser la regex `chien|chat`. Comme précédemment, perl essayera de reconnaître la regex le plus tôt possible dans la chaîne. À chaque position, perl essayera la première possibilité : `chien`. Si `chien` ne correspond pas, perl essayera la possibilité suivante : `chat`. Si `chat` ne convient pas non plus alors il n'y a pas correspondance et perl se déplace à la position suivante dans la chaîne. Quelques exemples :

```
"chiens et chats" =~ /chien|chat|rat/; # reconnaît "chien"
"chiens et chats" =~ /chat|chien|rat/; # reconnaît "chien"
```

Bien que `chat` soit la première possibilité dans la seconde regex, `chien` est reconnue plus tôt dans la chaîne.

```
"chat"          =~ /c|ch|cha|chat/; # reconnaît "c"
"chat"          =~ /chat|cha|ch|c/; # reconnaît "chat"
```

En une position donnée, la possibilité qui est retenue est la première qui permet la reconnaissance de l'expression. Ici, toute les possibilités correspondent dès le premier caractère donc c'est la première qui est retenue.

7.2.4 Groupement et hiérarchie

Les meta-caractères de regroupement `()` permettent de traiter une partie d'une regex comme une seule entité. Une partie d'une regex est groupée en l'entourant de parenthèses. L'expression `para(pluie|chute)` peut reconnaître `para` suivi soit de `pluie` soit de `chute`. D'autres exemples :

```
/(a|b)b/;        # reconnaît 'ab' ou 'bb'
/^(a|b)c/;        # reconnaît 'ac' au début de la chaîne ou 'bc' n'importe où

/chat(on|)/;      # reconnaît 'chaton' ou 'chat'.
/chat(on(s|)|)/; # reconnaît 'chatons' ou 'chaton' ou 'chat'.
                  # Notez que les groupes peuvent être imbriqués

"20" =~ /(19|20|)\d\d/; # reconnaît la possibilité vide '()\d\d',
                       # puisque '20\d\d' ne peut pas correspondre
```

7.2.5 Mémorisation de la correspondance

Les meta-caractères de regroupement `()` permettent aussi la mémorisation de la partie reconnue de la chaîne. Pour chaque groupe, la partie reconnue de la chaîne va dans une variable spéciale `$1` ou `$2`, etc. Elles peuvent être utilisées comme des variables ordinaires :

```
# extraction des heures, minutes, secondes
$temps =~ /(\d\d):(\d\d):(\d\d)/; # reconnaît le format hh:mm:ss
$heures = $1;
$minutes = $2;
$secondes = $3;
```

Dans un contexte de liste, une mise en correspondance `/regex/` avec regroupement retourne la liste des valeurs (`$1`, `$2`, ...). Nous pouvons donc écrire :

```
($heures, $minutes, $secondes) = ($temps =~ /(\d\d):(\d\d):(\d\d)/);
```

Si les regroupements sont imbriqués, `$1` sera le groupe ayant la parenthèse ouvrante la plus à gauche, `$2` celui ayant la parenthèse ouvrante suivante, etc. Voici une expression rationnelle complexe avec les numéros des variables de groupes indiqués au-dessous :

```
/(ab(cd|ef)((gi)|j))/;
  1 2      34
```

Les références arrières `\1`, `\2...` sont associées aux variables `$1`, `$2...` Les références arrières sont utilisées *dans* l'expression rationnelle elle-même :

```
/(\w\w\w)\s\1/; # trouve les séquences telles que 'les les' dans la chaîne
```

`$1`, `$2...` ne devraient être utilisé qu'à l'extérieur de l'expression tandis que `\1`, `\2` ne devraient l'être qu'à l'intérieur.

7.2.6 Répétitions et quantificateurs

Les meta-caractères ou quantificateurs `?`, `*` et `{}` nous permettent de fixer le nombre de répétitions d'une portion de regex. Un quantificateur est placé juste après le caractère, la classe de caractères ou le regroupement à répéter. Ils ont le sens suivant :

- `a?` : reconnaît 'a' zéro ou une fois.
- `a*` : reconnaît 'a' zéro fois ou plus.
- `a+` : reconnaît 'a' au moins une fois.
- `a{n,m}` : reconnaît 'a' au moins `n` fois, mais pas plus de `m` fois.
- `a{n,}` : reconnaît 'a' au moins `n` fois.
- `a{n}` : reconnaît 'a' exactement `n` fois.

Voici quelques exemples :

```
/[a-z]+\s+\d*/; # reconnaît un mot en minuscules suivi d'au moins un blanc
                # et éventuellement d'un certain nombre de chiffres
/(\w+)\s+\1/;  # reconnaît la répétition d'un mot de longueur quelconque
$annee =~ /\d{2,4}/; # s'assure que l'année contient au moins 2 chiffres et
                    # pas plus de 4 chiffres
$annee =~ /\d{4}|\d{2}/; # meilleure reconnaissance; exclut le cas de 3 chiffres
```

Ces quantificateurs essayent d'entrer en correspondance avec la chaîne la plus longue possible tout en permettant à la regex d'être reconnue (ils sont gourmands). Donc :

```
$x = 'Ce chien est le mien';
$x =~ /^(.*) (ien) (.*)$/; # correspondance,
                          # $1 = 'Ce chien est le m'
                          # $2 = 'ien'
                          # $3 = '' (aucun caractère)
```

Le premier quantificateur `.*` consomme la chaîne la plus longue possible tout en laissant une possibilité de correspondance pour la regex globale. Le second quantificateur `.*` n'a plus de caractère disponible donc il est reconnu zéro fois.

7.2.7 Plus de correspondances

Il y a encore quelques détails que vous devez connaître à propos des opérateurs de correspondance. Dans le code

```
$motif = 'Seuss';
while (<>) {
    print if /$motif/;
}
```

perl doit réévaluer \$motif à chaque passage dans la boucle. Si \$motif ne change jamais, utilisez le modificateur //o pour n'effectuer qu'une seule fois l'interpolation. Si vous ne voulez aucune interpolation, utilisez les délimiteurs spéciaux m" :

```
@motif = ('Seuss');
m/@motif/; # reconnaît 'Seuss'
m'@motif'; # reconnaît la chaîne littérale '@motif'
```

Le modificateur //g demande à l'opérateur de mise en correspondance de s'appliquer à une chaîne autant de fois que possible. Dans un contexte scalaire, une recherche de correspondance assortie du modificateur //g sautera de reconnaissance en reconnaissance en se souvenant à chaque fois de l'endroit où elle s'est arrêtée la fois précédente. Vous pouvez récupérer la position atteinte via la fonction pos(). Par exemple :

```
$x = "chien chat maison"; # 3 mots
while ($x =~ /(\w+)/g) {
    print "le mot $1 se termine en ", pos $x, "\n";
}
```

affiche

```
le mot chien se termine en 5
le mot chat se termine en 10
le mot maison se termine en 17
```

L'échec de la reconnaissance ou la modification de la chaîne réinitialise la position. Si vous ne voulez pas de réinitialisation en cas d'échec, ajoutez le modificateur //c comme dans /regex/gc.

Dans un contexte de liste, //g retournera la liste complète des groupes reconnus ou, si il n'y a pas de regroupement, la liste des sous-chaînes reconnues par la regex complète. Donc :

```
@mots = ($x =~ /(\w+)/g); # correspondance,
# $mots[0] = 'chien'
# $mots[1] = 'chat'
# $mots[2] = 'maison'
```

7.2.8 Recherche et remplacement

On effectue une recherche et remplacement en utilisant s/regex/remplacement/modificateurs. remplacement est comme une chaîne Perl entre guillemets qui remplacera dans la chaîne la partie reconnue par la regex. Là aussi, l'opérateur =~ permet de choisir à quelle chaîne sera appliquée s///. Si s/// doit s'appliquer à \$_, il est possible d'omettre \$_ =~. S'il y a une correspondance, s/// retourne le nombre de remplacements effectués sinon il retourne faux. Voici quelques exemples :

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contient "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^(.*)'$/\1/; # supprime les apostrophes,
# $y contient "quoted words"
```

Lorsqu'on utilise l'opérateur s///, les variables \$1, \$2, etc. sont directement utilisables dans l'expression de remplacement. Avec le modificateur s///g, la recherche et remplacement auront lieu sur toutes les occurrences de l'expression rationnelle :

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # $x contient "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g; # $x contient "I batted four for four"
```

Le modificateur d'évaluation `s///e` ajoute un `eval{...}` autour de la chaîne de remplacement et c'est le résultat de cette évaluation qui sera substitué à la sous-chaîne reconnue. Quelques exemples :

```
# inverser tous les mots d'une chaîne
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge; # $x contient "eht tac ni eht tah"

# convertir un pourcentage en fraction
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e; # $x contient "A 0.39 hit rate"
```

Le dernier exemple montre que `s///` peut utiliser d'autres délimiteurs tels que `s!!!` ou `s{ }{ }...` ou même `s{ }//`. Si les délimiteurs sont des apostrophes `s''` alors l'expression rationnelle et la chaîne de remplacement sont considérées comme des chaînes entre apostrophes (pas d'interpolation des variables).

7.2.9 L'opérateur de découpage : split

`split /regex/, chaîne` découpe `chaîne` en une liste de sous-chaînes et retourne cette liste. La `regex` détermine la séquence de caractères à utiliser comme séparateur lors du découpage de `string`. Par exemple, pour découper une chaîne en mots, utilisez :

```
$x = "Calvin and Hobbes";
@mots = split /\s+/, $x; # $mots[0] = 'Calvin'
                        # $mots[1] = 'and'
                        # $mots[2] = 'Hobbes'
```

Pour extraire une liste de nombres séparés par des points-virgules :

```
$x = "1,618;2,718; 3,142";
@const = split /;\s*/, $x; # $const[0] = '1,618'
                          # $const[1] = '2,718'
                          # $const[2] = '3,142'
```

Si vous utilisez l'expression rationnelle vide `//`, la chaîne est découpée en ses caractères. Si l'expression rationnelle contient des regroupements alors la liste produite contiendra aussi les groupes reconnus :

```
$x = "/usr/bin";
@parts = split m!(/)! , $x; # $parts[0] = ''
                           # $parts[1] = '/'
                           # $parts[2] = 'usr'
                           # $parts[3] = '/'
                           # $parts[4] = 'bin'
```

Puisque le premier caractère de `$x` est reconnu comme délimiteur, `split` ajoute un élément vide au début de la liste.

7.3 BUGS

Aucun.

7.4 VOIR AUSSI

C'est un simple guide d'introduction. Pour un tutoriel plus complet voir *perlretut* et pour une référence complète voir *perltre*.

7.5 AUTEUR ET COPYRIGHT

Copyright (c) 2000 Mark Kvale All rights reserved.

This document may be distributed under the same terms as Perl itself.

Tous droits réservés.

Ce document peut être distribuer sous les mêmes termes que Perl.

7.5.1 Remerciements

L'auteur tient à remercier Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes et Mike Giroux pour leurs commentaires précieux.

7.6 TRADUCTION

7.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

7.6.2 Traducteur

Paul Gaborit (Paul.Gaborit at enstimac.fr).

7.6.3 Relecture

Aucune pour l'instant.

Chapitre 8

perlretut

Tutoriel des expressions rationnelles en Perl

8.1 DESCRIPTION

Remarque sur la traduction : on emploie couramment le terme "expression régulière" car le terme anglais est "regular expression" qui s'abrège en "regex". Mais ne nous y trompons pas, en français, ce sont bien des "expressions rationnelles".

Ce document propose un tutoriel dans le but de comprendre, créer et utiliser des expressions rationnelles en Perl. Il sert de complément à la documentation de référence sur les expressions rationnelles, *perlre*. Les expressions rationnelles font partie intégrante des opérateurs `m//`, `s///`, `qr//` et `split`, donc ce tutoriel a aussi des recoupements avec Opérateurs d'expression rationnelle in *perlop* et `split` in *perlfunc*.

Perl est largement reconnu pour ses capacités de manipulation de textes et les expressions rationnelles y sont pour beaucoup. Les expressions rationnelles en Perl permettent une flexibilité et une efficacité inconnues dans la plupart des autres langages. La maîtrise des expressions rationnelles même les plus simples vous permettra de manipuler du texte avec une surprenante facilité.

Qu'est-ce qu'une expression rationnelle ? Une expression rationnelle est tout simplement une chaîne de caractères qui décrit un motif. La notion de motif est couramment utilisée de nos jours. Par exemple, les motifs utilisés par un moteur de recherche pour trouver des pages web ou les motifs utilisés pour lister les fichiers dans un répertoire, e.g. `ls *.txt` ou `dir *.*`. En Perl, les motifs d'expressions rationnelles sont utilisés pour chercher dans des chaînes de caractères, pour extraire certaines parties d'une chaîne et pour réaliser des opérations de recherche et de remplacement.

Les expressions rationnelles ont la réputation d'être abstraites et difficiles à comprendre. Les expressions rationnelles sont construites par assemblage de concepts simples tels que des conditions et des boucles qui ne sont pas plus compliquées à comprendre que les conditions `if` et les boucles `while` du langage Perl lui-même. En fait, le véritable enjeu dans l'apprentissage des expressions rationnelles réside dans la compréhension de la notation laconique utilisée pour exprimer ces concepts.

Ce tutoriel aplanit la courbe d'apprentissage en présentant les concepts des expressions rationnelles, ainsi que leur notation, un par un et accompagnés d'exemples. La première partie de ce tutoriel commence par la simple recherche de mots pour aboutir aux concepts de base des expressions rationnelles. Si vous maîtrisez cette première partie, vous aurez tous les outils nécessaires pour résoudre 98% de vos besoins. La seconde partie de ce tutoriel est destinée à ceux qui sont déjà à l'aise avec les bases et qui recherchent des outils plus puissants. Elle explique les opérateurs les plus avancés des expressions rationnelles ainsi que les dernières innovations de la version 5.6.0.

Remarque : pour gagner du temps, 'expression rationnelle' est parfois abrégée par `regex` ou `regx`. `Regex` est plus naturel (pour un anglophone) que `regx` mais est aussi plus dur à prononcer (toujours pour un anglophone). Dans la documentation Perl, on oscille entre `regex` et `regx` ; en Perl, il y a toujours plusieurs façons d'abrégier. Dans ce tutoriel en français, nous n'utiliserons que rarement `regex` (N.d.t: même si l'abréviation française donne `exrpat` !).

8.2 Partie 1: les bases

8.2.1 Reconnaissance d'un mot simple

L'expression rationnelle la plus simple est un simple mot ou, plus généralement, une chaîne de caractères. Une expression rationnelle constituée d'un mot reconnaît toutes les chaînes qui contiennent ce mot :

```
"Hello World" =~ /World/; # est reconnu
```

Que signifie cette instruction Perl? "Hello World" est une simple chaîne de caractères entre guillemets. World est l'expression rationnelle et les // qui l'entourent (/World/) demandent à Perl d'en chercher une correspondance dans une chaîne. L'opérateur =~ associe la chaîne avec l'expression rationnelle de recherche de correspondance et produit une valeur vraie s'il y a correspondance ou faux sinon. Dans notre cas, World correspond au second mot dans "Hello World" et donc l'expression est vraie. De telles expressions sont pratique dans des conditions :

```
if ("Hello World" =~ /World/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}
```

Il existe de nombreuses variations utiles sur ce thème. Le sens de la correspondance peut-être inversée en utilisant l'opérateur !~ :

```
if ("Hello World" !~ /World/) {
    print "Il n'y a pas correspondance\n";
}
else {
    print "Il y a correspondance\n";
}
```

La chaîne littérale dans l'expression rationnelle peut être remplacée par une variable :

```
$greeting = "World";
if ("Hello World" =~ /$greeting/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}
```

Si vous recherchez dans la variable spéciale par défaut \$_, la partie \$_ =~ peut être omise :

```
$_ = "Hello World";
if (/World/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}
```

Et finalement, les délimiteurs par défaut // pour une recherche de correspondance peuvent être remplacés par n'importe quels autres délimiteurs en les préfixant par un 'm' :

```
"Hello World" =~ m!World!; # correspond, délimiteur '!'
"Hello World" =~ m{World}; # correspond, notez le couple '{}
"/usr/bin/perl" =~ m"/perl"; # correspond après '/usr/bin'
# '/' devient un caractère comme un autre
```

/World/, m!World! et m{World} représentent tous la même chose. Lorsque, par exemple, des guillemets (") sont utilisés comme délimiteurs, la barre oblique '/' devient un caractère ordinaire et peut être utilisée dans une expression rationnelle sans problème.

Regardons maintenant comment différentes expressions rationnelles peuvent ou non trouver une correspondance dans "Hello World" :

```
"Hello World" =~ /world/; # ne correspond pas
"Hello World" =~ /o W/;   # correspond
"Hello World" =~ /oW/;   # ne correspond pas
"Hello World" =~ /World /; # ne correspond pas
```

La première expression rationnelle `world` ne correspond pas car les expressions rationnelles sont sensibles à la casse. La deuxième expression rationnelle trouve une correspondance car la sous-chaîne `'o W'` apparaît dans la chaîne `"Hello World"`. Le caractère espace `' '` est traité comme n'importe quel autre caractère dans une expression rationnelle et il est nécessaire ici pour trouver une correspondance. L'absence du caractère espace explique la non-reconnaissance de la troisième expression rationnelle. La quatrième expression rationnelle `'World '` ne trouve pas de correspondance car il y a un espace à la fin de l'expression rationnelle et non à la fin de la chaîne. La leçon à tirer de ces exemples est qu'une expression rationnelle doit correspondre *exactement* à une partie de la chaîne pour être reconnue.

Si une expression rationnelle peut être reconnue à plusieurs endroits dans une chaîne, Perl choisira toujours la correspondance au plus tôt :

```
"Hello World" =~ /o/;      # correspond au 'o' dans 'Hello'
"That hat is red" =~ /hat/; # correspond au 'hat' dans 'That'
```

Il y a encore quelques points que vous devez savoir à propos de la reconnaissance de caractères. En premier lieu, tous les caractères ne peuvent pas être utilisés tels quels pour une correspondance. Quelques caractères, appelés *meta-caractères*, sont réservés pour des notations d'expressions rationnelles. Les meta-caractères sont :

```
{ } [ ] ( ) ^ $ . | * + ? \
```

La signification de chacun d'eux sera expliquée plus loin dans ce tutoriel. Pour l'heure, il vous suffira de savoir qu'un meta-caractère sera recherché tel quel si vous le précédez d'un backslash (une barre oblique inversée) :

```
"2+2=4" =~ /2+2/;      # pas de correspondance, + est un meta-caractère
"2+2=4" =~ /2\+2/;    # correspond, \+ est traité comme un + ordinaire
"The interval is [0,1)."
```

```
 =~ /[0,1)./          # c'est une erreur de syntaxe !
"The interval is [0,1)."
```

```
 =~ /\[0,1)\./       # correspond
"#!/usr/bin/perl" =~ /#!\usr\bin\perl/; # correspond
```

Dans la dernière expression rationnelle, les slash `'/'` sont aussi précédés d'un backslash parce que le slash est utilisé comme délimiteur de l'expression rationnelle. Par contre, cela peut aboutir au LTS (leaning toothpick syndrome) et il est donc souvent préférable de changer de délimiteur.

```
"#!/usr/bin/perl" =~ m#!/usr/bin/perl!; # plus facile à lire
```

Le caractère backslash `'\'` est lui-même un meta-caractère et doit donc être backslashé (précédé d'un backslash) :

```
'C:\WIN32' =~ /C:\\WIN/; # correspond
```

En plus des meta-caractères, il y a quelques caractères ASCII qui n'ont pas d'équivalent affichable et sont donc représentés par des *séquences d'échappement*. Les plus courants sont `\t` pour une tabulation, `\n` pour un saut de ligne, `\r` pour un retour chariot et `\a` pour un beep. Si votre chaîne se présente plutôt comme une séquence d'octets quelconques, les séquences d'échappement en octal, telle que `\033`, ou en hexadécimal, telle que `\x1B` seront peut-être une représentation plus naturelle pour vos octets. Voici quelques exemples d'utilisation des séquences d'échappement :

```
"1000\t2000" =~ m(0\t2) # correspond
"1000\n2000" =~ /0\n20/  # correspond
"1000\t2000" =~ /\000\t2/ # ne correspond pas, "0" ne "\000"
"cat" =~ /\143\x61\x74/  # correspond, une façon bizarre d'épeler 'cat'
```

Si vous pratiquez déjà Perl, tout cela doit vous sembler familier. Des séquences similaires sont utilisées dans les chaînes entre guillemets. De fait, les expressions rationnelles en Perl sont traitées comme des chaînes entre guillemets. Cela signifie que l'on peut utiliser des variables dans les expressions rationnelles. Exactement comme pour les chaînes entre guillemets, chaque variable sera remplacée par sa valeur avant que l'expression rationnelle soit utilisée à la recherche de correspondances. Donc :

```
$foo = 'house';
'housecat' =~ /$foo/;      # correspond
'cathouse' =~ /cat$foo/;  # correspond
'housecat' =~ /${foo}cat/; # correspond
```

Jusqu'ici, ça va. Avec les connaissances qui précèdent vous pouvez déjà rechercher toutes les chaînes littérales imaginables. Voici une émulation *très simple* du programme Unix `grep` :

```
% cat > simple_grep
#!/usr/bin/perl
$regexp = shift;
while (<>) {
    print if /$regexp/;
}
^D

% chmod +x simple_grep

% simple_grep abba /usr/dict/words
Babbage
cabbage
cabbages
sabbath
Sabbathize
Sabbathizes
sabbatical
scabbard
scabbards
```

Ce programme est très simple à comprendre. `#!/usr/bin/perl` est le moyen standard pour invoquer perl. `$regexp = shift;` mémorise le premier argument de la ligne de commande en tant qu'expression rationnelle à utiliser et laisse le reste des arguments pour qu'ils soient traités comme des fichiers. `while (<>)` parcourt toutes les lignes de tous les fichiers. Pour chaque ligne, `print if /$regexp/;` affiche la ligne si l'expression rationnelle trouve une correspondance dans la ligne. Dans cette ligne, `print` et `/$regexp/` utilisent implicitement tous les deux la variable par défaut `$_`.

Dans toutes les expressions rationnelles précédentes, si l'expression rationnelle trouvait une correspondance n'importe où dans la chaîne, on considérerait qu'elle correspondait. Parfois, par contre, nous aimerions spécifier *l'endroit* dans la chaîne où l'expression rationnelle doit trouver une correspondance. Pour cela, nous devons utiliser les meta-caractères d'ancrage `^` et `$`. L'ancre `^` demande à correspondre au début de la chaîne et l'ancre `$` demande à correspondre à la fin de la chaîne ou juste avant le passage à la ligne avant la fin de la chaîne. Voici comment elles sont utilisées :

```
"housekeeper" =~ /keeper/;      # correspond
"housekeeper" =~ /^keeper/;    # ne correspond pas
"housekeeper" =~ /keeper$/;    # correspond
"housekeeper\n" =~ /keeper$/;  # correspond
```

La deuxième expression rationnelle ne correspond pas parce que `^` contraint `keeper` à ne correspondre qu'au début de la chaîne. Or `"housekeeper"` contient un `"keeper"` qui débute au milieu de la chaîne. La troisième expression rationnelle correspond puisque le `$` contraint `keeper` à n'être reconnue qu'à la fin de la chaîne.

Lorsque `^` et `$` sont utilisés ensemble, l'expression rationnelle doit être ancrée à la fois au début et à la fin de la chaîne, i.e., l'expression rationnelle correspond donc à la chaîne entière. Considérons :

```
"keeper" =~ /^keep$/;          # ne correspond pas
"keeper" =~ /^keeper$/;       # correspond
""          =~ /^$/;           # ^$ correspond à la chaîne vide
```

La première expression rationnelle ne peut pas correspondre puisque la chaîne contient plus que `keep`. Puisque la deuxième expression rationnelle est exactement la chaîne, elle correspond. L'utilisation combinée de `^` et de `$` force la chaîne entière à correspondre et vous donne donc un contrôle complet sur les chaînes qui correspondent et celles qui ne correspondent pas. Supposons que vous cherchez un individu nommé `bert` :

```
"dogbert" =~ /bert/; # correspond, mais ce n'est pas ce qu'on cherche

"dilbert" =~ /^bert/; # ne correspond pas mais...
"bertram" =~ /^bert/; # correspond, ce n'est donc pas encore bon

"bertram" =~ /^bert$/; # ne correspond pas, ok
"dilbert" =~ /^bert$/; # ne correspond pas, ok
"bert"    =~ /^bert$/; # correspond, parfait
```

Bien sûr, dans le cas d'une chaîne littérale, n'importe qui utiliserait l'opérateur d'égalité des chaînes `$string eq 'bert'` et cela serait beaucoup plus efficace. L'expression rationnelle `^...$` devient beaucoup plus pratique lorsqu'on lui ajoute la puissance des outils d'expression rationnelle que nous verrons plus bas.

8.2.2 Utilisation des classes de caractères

Bien que certains puissent se satisfaire des expressions rationnelles précédentes qui ne reconnaissent que des chaînes littérales, nous n'avons qu'effleuré la technologie des expressions rationnelles. Dans cette section et les suivantes nous allons présenter les concepts d'expressions rationnelles (et les meta-caractères associés) qui permettent à une expression rationnelle de représenter non seulement une seule séquence de caractères mais aussi *toute une classe* de séquences.

L'un de ces concepts est celui de *classe de caractères*. Une classe de caractères autorise un ensemble de caractères, plutôt qu'un seul caractère, à correspondre en un point particulier de l'expression rationnelle. Les classes de caractères sont entourées de crochets [...] avec l'ensemble de caractères placé à l'intérieur. Voici quelques exemples :

```
/cat/; # reconnaît 'cat'
/[bcr]at/; # reconnaît 'bat', 'cat', ou 'rat'
/item[0123456789]/; # reconnaît 'item0' ou ... ou 'item9'
"abc" =~ /[cab]/; # reconnaît le 'a'
```

Dans la dernière instruction, bien que 'c' soit le premier caractère dans la classe, c'est le 'a' qui correspond car c'est le caractère le plus au début de la chaîne avec lequel l'expression rationnelle peut correspondre.

```
/[oO][uU][iI]/; # reconnaît 'oui' sans tenir compte de la casse
# 'oui', 'Oui', 'OUI', etc.
```

Cette expression rationnelle réalise une tâche courante : une recherche de correspondance insensible à la casse. Perl fournit un moyen d'éviter tous ces crochets en ajoutant simplement un 'i' après l'expression rationnelle. Donc `/[oO][uU][iI]/;` peut être réécrit en `/oui/i;`. Le 'i' signifie insensible à la casse et est un exemple de **modificateur** de l'opération de reconnaissance. Nous rencontrerons d'autres modificateurs plus tard dans ce tutoriel.

Nous avons vu dans la section précédente qu'il y avait des caractères ordinaires, qui se représentaient eux-mêmes et des caractères spéciaux qui devaient être backslashés pour se représenter. C'est la même chose dans les classes de caractères mais les ensembles de caractères ordinaires et spéciaux ne sont pas les mêmes. Les caractères spéciaux dans une classe de caractères sont `]` `^` `$` (et les délimiteurs, quels qu'ils soient). `]` est spécial car il indique la fin de la classe de caractères. `$` est spécial car il indique une variable scalaire. `\` est spécial car il est utilisé pour les séquences d'échappement comme précédemment. Voici comment les caractères spéciaux `]` `$` `\` sont utilisés :

```
/[\]c]def/; # reconnaît 'def' ou 'cdef'
$x = 'bcr';
/[$x]at/; # reconnaît 'bat', 'cat', ou 'rat'
/[\\$x]at/; # reconnaît '$at' ou 'xat'
/[\\\$x]at/; # reconnaît '\at', 'bat', 'cat', ou 'rat'
```

Les deux derniers exemples sont un peu complexes. Dans `[\$x]`, le backslash protège le signe dollar et donc la classe de caractères contient deux membres : `$` et `x`. Dans `[\\$x]`, le backslash est lui-même protégé et donc `$x` est traité comme une variable à laquelle on substitue sa valeur comme dans les chaînes entre guillemets.

Le caractère spécial `-` agit comme un opérateur d'intervalle dans une classe de caractères et donc un ensemble de caractères contigus peut être décrit comme un intervalle. Grâce aux intervalles, les classes peu maniables comme `[0123456789]` et `[abc...xyz]` deviennent très simples : `[0-9]` et `[a-z]`. Quelques exemples :

```

/item[0-9]/; # reconnaît 'item0' ou ... ou 'item9'
/[0-9bx-z]aa/; # reconnaît '0aa', ..., '9aa',
                # 'baa', 'xaa', 'yaa', or 'zaa'
/[0-9a-fA-F]/; # reconnaît un chiffre hexadécimal
/[0-9a-zA-Z_]/; # reconnaît un caractère d'un "mot",
                # comme ceux qui composent les noms en Perl

```

Si '-' est le premier ou le dernier des caractères dans une classe, il est traité comme un caractère ordinaire; [-ab], [ab-] et [a\b] sont équivalents.

Le caractère spécial ^ en première position d'une classe de caractères indique une *classe de caractères inverse* qui reconnaît n'importe quel caractère sauf ceux présents entre les crochets. Dans les deux cas, [...] et [^...], il faut qu'un caractère soit reconnu sinon la reconnaissance échoue. Donc :

```

/[^a]at/; # ne reconnaît pas 'aat' ou 'at', mais reconnaît
           # tous les autres 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # reconnaît un caractère non numérique
/[a^]at/; # reconnaît 'aat' or '^at'; ici '^' est ordinaire

```

Même [0-9] peut être ennuyeux à écrire plusieurs fois. Donc pour minimiser la frappe et rendre les expressions rationnelles plus lisibles, Perl propose plusieurs abréviations pour les classes les plus communes :

- \d reconnaît un chiffre, non seulement [0-9] mais aussi les chiffres des langues non-romaines.
- \s reconnaît un caractère d'espace, l'ensemble [\ \t\r\n\f] et d'autres caractères.
- \w reconnaît un caractère de mot (alphanumérique ou _), non seulement [0-9a-zA-Z_] mais aussi les caractères des langues non-romaines.
- \D est la négation de \d; il représente n'importe quel caractère sauf un chiffre, autrement dit [^\d].
- \S est la négation de \s; il représente n'importe quel caractère qui ne soit pas d'espace [^\s].
- \W est la négation de \w; il représente n'importe quel caractère qui ne soit pas un caractère de mot [^\w].
- Le point '.' reconnaît n'importe quel caractère sauf "\n" (à moins que le modificateur //s soit activé comme expliqué plus bas).

Les abréviations \d\s\w\D\S\W peuvent être utilisées à l'intérieur ou à l'extérieur des classes de caractères. Voici quelques exemples :

```

/\d\d:\d\d:\d\d/; # reconnaît une heure au format hh:mm:ss
/[\d\s]/;         # reconnaît n'importe quel chiffre ou espace
/\w\W\w/;        # reconnaît un caractère mot, suivi d'un
                 # caractère non-mot, suivi d'un caractère mot
/..rt/;          # reconnaît deux caractères, suivis de 'rt'
/end\./;         # reconnaît 'end.'
/end[.]/;        # la même chose, reconnaît 'end.'

```

Puisque un point est un meta-caractère, il doit être backslashé pour reconnaître un point ordinaire. Puisque, par exemple, \d et \w sont des ensembles de caractères, il est incorrect de penser que [^\d\w] est équivalent à [\D\W]; en fait [^\d\w] est la même chose que [^\w], qui est la même chose que [\W]. C'est l'application des lois ensemblistes de De Morgan.

Une ancre pratique dans les expressions rationnelles de base est l'*ancrage de mot* \b. Elle reconnaît la frontière entre un caractère mot et un caractère non-mot \w\W ou \W\w :

```

$x = "Housecat catenates house and cat";
$x =~ /cat/; # reconnaît cat dans 'housecat'
$x =~ /\bcat/; # reconnaît cat dans 'catenates'
$x =~ /cat\b/; # reconnaît cat dans 'housecat'
$x =~ /\bcat\b/; # reconnaît 'cat' à la fin de la chaîne

```

Notez que dans le dernier exemple, la fin de la chaîne est considérée comme une frontière de mot.

Vous devez vous demander pourquoi '.' reconnaît tous les caractères sauf "\n" - pourquoi pas tous les caractères ? C'est parce que le plus souvent on effectue des mises en correspondance par ligne et que nous voulons ignorer le caractère de passage à la ligne. Par exemple, bien que la chaîne "\n" représente une ligne, nous aimons la considérer comme vide. Par conséquent :

```

"" =~ /^$/; # est reconnue
"\n" =~ /^$/; # est reconnue; $ s'ancree avant "\n"

```

```

""    =~ /.;/;    # n'est pas reconnue; nécessite un caractère
""    =~ /^.$/;   # n'est pas reconnue; nécessite un caractère
"\n"  =~ /^.$/;   # n'est pas reconnue; nécessite un caractère autre que "\n"
"a"    =~ /^.$/;   # est reconnue
"a\n"  =~ /^.$/;   # est reconnue; $ s'ancre avant "\n"

```

Ce comportement est pratique parce qu'habituellement nous voulons ignorer les passages à la ligne lorsque nous mettons en correspondance les caractères d'une ligne. Parfois, en revanche, nous voulons tenir compte des passages à la ligne. Nous pouvons aussi vouloir que les ancres `^` et `$` puissent s'ancre au début et à la fin des lignes plutôt que simplement au début et à la fin de la chaîne. Perl nous permet de choisir entre ignorer ou tenir compte des passages à la ligne grâce aux modificateurs `//s` et `//m`. `//s` et `//m` signifient ligne simple et multi-ligne et ils déterminent si une chaîne doit être considérée comme une chaîne continue ou comme un ensemble de lignes. Les deux modificateurs agissent sur deux aspects de l'interprétation de l'expression rationnelle : 1) comment la classe de caractères `'.'` est définie et 2) où les ancres `^` et `$` peuvent s'ancre. Voici les quatre combinaisons :

- pas de modificateurs (`//`) : comportement par défaut. `'.'` reconnaît n'importe quel caractère sauf `"\n"`. `^` correspond uniquement au début de la chaîne et `$` correspond uniquement à la fin de la chaîne ou avant le passage à la ligne avant la fin de la chaîne.
- modificateur `s` (`//s`) : traite la chaîne comme une seule longue ligne. `'.'` reconnaît tous les caractères, même `"\n"`. `^` correspond uniquement au début de la chaîne et `$` correspond uniquement à la fin de la chaîne ou avant le passage à la ligne avant la fin de la chaîne.
- modificateur `m` (`//m`) : traite la chaîne comme un ensemble de lignes. `'.'` reconnaît n'importe quel caractère sauf `"\n"`. `^` et `$` peuvent correspondre au début et à la fin de n'importe quelle ligne dans la chaîne.
- les deux modificateurs `s` et `m` (`//sm`) : traite la chaîne comme une seule longue ligne mais détecte les lignes multiples. `'.'` reconnaît tous les caractères, même `"\n"`. `^` et `$` peuvent correspondre au début et à la fin de n'importe quelle ligne dans la chaîne.

Voici des exemples d'utilisation de `//s` et `//m` :

```

$x = "There once was a girl\nWho programmed in Perl\n";

$x =~ /^Who/;    # non reconnue, "Who" n'est pas en début de chaîne
$x =~ /^Who/s;   # non reconnue, "Who" n'est pas en début de chaîne
$x =~ /^Who/m;   # reconnue, "Who" au début de la seconde ligne
$x =~ /^Who/sm;  # reconnue, "Who" au début de la seconde ligne

$x =~ /girl.Who/; # non reconnue, le "." ne reconnaît pas "\n"
$x =~ /girl.Who/s; # reconnue, le "." reconnaît "\n"
$x =~ /girl.Who/m; # non reconnue, le "." ne reconnaît pas "\n"
$x =~ /girl.Who/sm; # reconnue, le "." reconnaît "\n"

```

La plupart du temps, le comportement par défaut est ce que nous voulons mais `//s` et `//m` sont occasionnellement très utiles. Si `//m` est utilisé, le début de la chaîne peut encore être reconnu par `\A` et la fin de la chaîne peut aussi être reconnue soit par l'ancre `\Z` (qui est reconnue à la fin de la chaîne ou juste avant le passage à la ligne, comme `$`) soit par l'ancre `\z` (qui n'est reconnue qu'à la fin de la chaîne) :

```

$x =~ /^Who/m;    # reconnue, "Who" au début de la seconde ligne
$x =~ /\AWho/m;   # non reconnue, "Who" n'est pas au début de la chaîne

$x =~ /girl$/m;   # reconnue, "girl" à la fin de la première ligne
$x =~ /girl\Z/m;  # non reconnue, "girl" n'est pas à la fin de la chaîne

$x =~ /Perl\Z/m;  # reconnue, "Perl" est juste avant le passage à la ligne
                  # de la fin de chaîne
$x =~ /Perl\z/m;  # non reconnue, "Perl" n'est pas à la fin de la chaîne

```

Nous savons maintenant comment créer des choix à travers des classes de caractères dans les expressions rationnelles. Qu'en est-il de choix entre des mots ou des chaînes de caractères ? Ce sont ces choix qui sont décrits dans la section suivante.

8.2.3 Reconnaître ceci ou cela

Parfois nous aimerions que notre expression rationnelle soit capable de reconnaître différents mots ou suites de caractères. Ceci s'effectue en utilisant le meta-caractère d'*alternative* |. Pour reconnaître dog ou cat, nous formons l'expression rationnelle dog|cat. Comme précédemment, Perl essaie de reconnaître l'expression rationnelle le plus tôt possible dans la chaîne. À chaque position, Perl essaie en premier de reconnaître la première branche de l'alternative, dog. Si dog n'est pas reconnu, Perl essaie ensuite la branche suivante, cat. Si cat n'est pas reconnu non plus alors la mise en correspondance échoue et Perl se déplace à la position suivante dans la chaîne. Quelques exemples :

```
"cats and dogs" =~ /cat|dog|bird/; # reconnaît "cat"
"cats and dogs" =~ /dog|cat|bird/; # reconnaît "cat"
```

Même si dog est la première branche de l'alternative dans le second exemple, cat est reconnu plus tôt dans la chaîne.

```
"cats"      =~ /c|ca|cat|cats/; # reconnaît "c"
"cats"      =~ /cats|cat|ca|c/; # reconnaît "cats"
```

Ici, toutes les branches peuvent correspondre à la première position dont la première branche reconnue est celle qui est retenue. Si certaines branches de l'alternative sont des troncatures des autres, placez les plus longues en premier pour leur donner une chance d'être reconnues.

```
"cab" =~ /a|b|c/ # reconnaît "c"
      # /a|b|c/ == /[abc]/
```

Ce dernier exemple montre que les classes de caractères sont comme des alternatives entre caractères. À une position donnée, la première branche qui permet une mise en correspondance de l'expression rationnelle est celle qui sera reconnue.

8.2.4 Regroupement et reconnaissance hiérarchique

Les alternatives permettent à une expression rationnelle de choisir entre différentes branches mais elles restent non satisfaisantes en elles-mêmes. Pour la simple raison que l'alternative est une expression rationnelle complète alors que parfois nous aimerions que ce ne soit qu'une partie de l'expression rationnelle. Par exemple, supposons que nous voulions reconnaître housecat ou housekeeper. L'expression rationnelle housecat|housekeeper fonctionne mais est inefficace puisque nous avons dû taper house deux fois. Il serait pratique d'avoir une partie de l'expression rationnelle qui soit constante, comme house, et une partie qui soit une alternative, comme cat|keeper.

Les meta-caractères de *regroupement* () résolvent ce problème. Le regroupement permet à une partie d'une expression rationnelle d'être traitée comme une seule entité. Une partie d'une expression rationnelle est regroupée en la plaçant entre des parenthèses. Donc nous pouvons résoudre le problème housecat|housekeeper en formant l'expression rationnelle house(cat|keeper). L'expression rationnelle house(cat|keeper) signifie cherche house suivi soit par cat soit par keeper. Quelques exemples :

```
/(a|b)b/;      # reconnaît 'ab' ou 'bb'
/(ac|b)b/;    # reconnaît 'acb' ou 'bb'
/^(a|b)c/;    # reconnaît 'ac' au début de la chaîne ou 'bc' n'importe où
/(a|[bc])d/;  # reconnaît 'ad', 'bd', ou 'cd'
```

```
/house(cat|)/; # reconnaît soit 'housecat' soit 'house'
/house(cat(s)|)/; # reconnaît soit 'housecats' soit 'housecat' soit
                  # 'house'. Les groupes peuvent être imbriqués.
```

```
/(19|20|)\d\d/; # reconnaît les années 19xx, 20xx, ou xx
"20" =~ /(19|20|)\d\d/; # reconnaît la branche vide '()\d\d',
                        # puisque '20\d\d' ne peut pas correspondre
```

Les alternatives se comportent de la même manière, qu'elles soient dans ou hors d'un groupe : à une position donnée, c'est la branche la plus à gauche qui est choisie tant qu'elle permet la reconnaissance de l'expression rationnelle. Donc dans notre dernier exemple, à la première position de la chaîne, "20" est reconnu par la deuxième branche de l'alternative mais il ne reste rien pour être reconnu par les deux chiffres suivants \d\d. Alors Perl essaie la branche suivante qui est la branche vide et ça marche puisque "20" est composé de deux chiffres.

Le processus d'essai d'une branche pour voir si elle convient puis de retour en arrière pour en essayer une autre est appelé *retour arrière* (*backtracking* en anglais). Les termes 'retour arrière' viennent de l'idée que la reconnaissance d'une expression rationnelle est comme une marche en forêt. La reconnaissance de l'expression rationnelle est comme l'arrivée à destination. Il y a plusieurs points de départ possibles, un pour chaque position dans la chaîne et chacun d'eux est essayé dans l'ordre, de gauche à droite. À partir de chaque point de départ, il y a plusieurs chemins dont certains sont des impasses et d'autres vous amènent à destination. Lorsque vous marchez sur un chemin et qu'il aboutit à une impasse, vous devez retourner en arrière pour essayer un autre chemin. Vous êtes persévérant et vous ne vous déclarez vaincu que lorsque vous avez essayé tous les chemins à partir de tous les points de départ sans aboutir à destination. Pour être plus concret, voici une analyse pas à pas de ce que Perl fait lorsqu'il essaye de reconnaître l'expression rationnelle :

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

1. On démarre avec la première lettre de la chaîne 'a'.
2. On essaie la première branche de la première alternative, le groupe 'abd'.
3. On reconnaît un 'a' suivi d'un 'b'. Jusqu'ici, ça va.
4. 'd' dans l'expression rationnelle ne correspond pas au 'c' dans la chaîne - une impasse. On effectue alors un retour arrière de deux caractères et on essaie la seconde branche de la première alternative, le groupe 'abc'.
5. On reconnaît un 'a' suivi d'un 'b' suivi d'un 'c'. Nous avons donc satisfait le premier regroupement. On positionne \$1 à 'abc'.
6. On continue avec la seconde alternative et on choisit la première branche 'df'.
7. On reconnaît le 'd'.
8. 'f' dans l'expression rationnelle ne correspond pas au 'e' dans la chaîne; c'est donc une impasse. Retour arrière d'un caractère et choix de la seconde branche de la seconde alternative 'd'.
9. 'd' est reconnu. Le second regroupement est satisfait et on positionne \$2 à 'd'.
10. Nous sommes à la fin de l'expression rationnelle. Nous sommes donc arrivés à destination ! Nous avons reconnu 'abcd' dans la chaîne "abcde".

Il y a deux choses à dire sur cette analyse. Tout d'abord, la troisième alternative ('de') dans le second regroupement permet aussi une reconnaissance, mais nous nous sommes arrêtés avant d'y arriver - à une position donnée, l'alternative la plus à gauche l'emporte. Ensuite, nous avons obtenu une reconnaissance au premier caractère ('a') de la chaîne. Si la reconnaissance n'avait pas été possible à cette première position, Perl se serait déplacé à la deuxième position ('b') pour essayer à nouveau une reconnaissance complète. C'est uniquement lorsque tous les chemins et toutes les positions ont été essayés sans succès que Perl s'arrête et déclare fausse l'assertion `$string =~ /(abd|abc)(df|d|de)/;`

Même avec tout ce boulot, les expressions rationnelles restent remarquablement rapides. Pour améliorer cela, Perl compile les expressions rationnelles en une séquence compacte d'opérations qui peut parfois tenir dans le cache du processeur. Lorsque le code est exécuté, ces opérations peuvent alors tourner à plein régime et chercher très rapidement.

8.2.5 Extraire ce qui est reconnu

Les meta-caractères de regroupement () servent aussi à une fonction totalement différente : ils permettent l'extraction des parties d'une chaîne qui ont trouvé une correspondance. C'est très pratique pour trouver ce qui a été reconnu et pour le traitement de texte en général. Pour chaque groupe, la partie qui a été reconnue est stockée dans les variables spéciales \$1, \$2, etc. Elles peuvent être utilisées comme des variables ordinaires :

```
# extraction des heures, minutes et secondes
if ($time =~ /(\d\d):(\d\d):(\d\d)/) { # reconnaît le format hh:mm:ss
    $heures = $1;
    $minutes = $2;
    $secondes = $3;
}
```

Pour l'instant, nous savons que, dans un contexte scalaire, `$time =~ /(\d\d):(\d\d):(\d\d)/` retourne une valeur vraie ou fausse. Dans un contexte de liste, en revanche, il retourne la liste de valeurs reconnues (\$1,\$2,\$3). Nous pouvons donc écrire du code plus compact :

```
# extraction des heures, minutes et secondes
($heures, $minutes, $secondes) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

Si les regroupements sont imbriqués dans l'expression rationnelle, \$1 recevra le groupe dont la parenthèse ouvrante est la plus à gauche, \$2 recevra le groupe dont la parenthèse ouvrante est la seconde la plus à gauche, etc. Par exemple, voici une expression rationnelle avec regroupements :

```
/(ab(cd|ef)((gi)|j))/;
 1 2      34
```

Si cette expression rationnelle est reconnue, \$1 contiendra une chaîne commençant par 'ab', \$2 contiendra soit 'cd' soit 'ef', \$3 sera soit 'gi' soit 'j' et finalement \$4 sera 'gi', exactement comme \$3, ou restera non défini.

De manière pratique, perl assigne à \$+ la chaîne associée au plus haut numéro des \$1, \$2... qui a été affectée (et \$^N sera la valeur des variables \$1, \$2... la plus récemment affectée, c'est-à-dire celle associée à la parenthèse fermante la plus à droite utilisée lors de la reconnaissance).

8.2.6 Les références arrières

Associées avec les variables \$1, \$2, ..., on trouve les *références arrières* : \1, \2, ... Les références arrières sont simplement des variables de reconnaissance qui peuvent être utilisées à l'intérieur de l'expression rationnelle. C'est une fonctionnalité vraiment sympathique - ce qui est reconnu plus tard dans une expression rationnelle dépend de ce qui a été reconnu plus tôt dans l'expression rationnelle. Supposons que nous voulons chercher les mots doublés dans un texte comme 'the the'. L'expression rationnelle suivante trouve tous les mots de trois lettres doublés avec un espace entre les deux :

```
/\b(\w\w\w)\s\1\b/;
```

Le regroupement affecte une valeur à \1 et donc la même séquence de 3 lettres est utilisée pour les deux parties. Voici quelques mots avec des parties répétées :

```
% simple_grep '^(\w\w\w\w|\w\w\w|\w\w|\w)\1$' /usr/dict/words
beriberi
booboo
coco
mama
murmur
papa
```

L'expression rationnelle commence par un regroupement qui considère d'abord les combinaisons de 4 lettres, puis de 3 lettres, etc., et utilise ensuite \1 pour chercher une répétition. Bien que \$1 et \1 représente la même chose, faites attention à n'utiliser les variables \$1, \$2, ... qu'à l'extérieur d'une expression rationnelle et à n'utiliser les références arrières \1, \2, ... qu'à l'intérieur d'une expression rationnelle. Ne pas y prêter attention peut amener à des résultats surprenants et indéfinis.

8.2.7 Références arrières relatives

Le comptage des parenthèses ouvrantes pour trouver le numéro correct attribué à une référence arrière est sujet à erreur en présence de plusieurs regroupements. Une technique plus pratique est proposé depuis Perl 5.10 : les références arrières relatives. Pour se référer au dernier regroupement qui précède on peut utiliser \g{-1}, l'avant-dernier regroupement est accessible via \g{-2} et ainsi de suite.

En plus d'améliorer lisibilité et la maintenabilité, un autre avantage des références arrières relatives est illustré pour l'exemple suivant où un simple motif est utilisé pour reconnaître quelques chaînes particulières :

```
$a99a = '([a-z])(\d)\2\1'; # reconnaît a11a, g22g, x33x, etc.
```

Maintenant que ce motif est stocké dans une chaîne, nous pouvons être tenté de l'utiliser à l'intérieur d'un autre motif :

```
$ligne = "code=e99e";
if ($ligne =~ /\^(w+)=a99a$/){ # comportement inattendu !
    print "$1 est valide\n";
} else {
    print "ligne invalide : '$ligne'\n";
}
```

Mais ça ne marche pas – tout du moins pas comme nous l'attendions. Ce n'est qu'en insérant par interpolation la valeur de \$a99a et en regardant l'expression rationnelle obtenue qu'on constate que les références arrières sont erronées – le regroupement (w+) a capté le numéro 1 et décalé d'un cran le regroupement présent dans \$a99a. Cela peut être évité en utilisant les références arrières relatives :

```
$a99a = '([a-z])(\d)\g{-1}\g{-2}'; # Ok pour être interpolé
```

8.2.8 Références arrières nommées

Perl 5.10 propose aussi les regroupements nommés et les références arrières nommées. Pour rattacher un nom à un groupe, vous pouvez écrire soit (`?<nom>...`) soit (`'nom'...`). La référence arrière correspondante s'écrit alors `\g{nom}`. Il est possible d'attacher le même nom à plusieurs regroupements mais seul le plus à gauche d'entre eux peut-être référencé. En dehors des expressions rationnelles, on peut accéder aux regroupements nommés via la table de hachage `%+`.

Supposons que vous cherchez à reconnaître des dates fournies dans l'un des trois formats suivants : `yyyy-mm-dd`, `mm/dd/yyyy` ou `dd.mm.yyyy`. Vous pouvez écrire trois motifs qui tous utiliseront les noms 'd', 'm' et 'y' associés aux regroupements reconnaissant chacun une composante de la date. L'opération de reconnaissance suivante combinent ces trois motifs dans une alternative :

```
$fmt1 = '(?<y>\d\d\d\d)-(?<m>\d\d)-(?<d>\d\d)';
$fmt2 = '(?<m>\d\d)/(?<d>\d\d)/(?<y>\d\d\d\d)';
$fmt3 = '(?<d>\d\d)\.(?<m>\d\d)\.(?<y>\d\d\d\d)';
for my $d qw( 2006-10-21 15.01.2007 10/31/2005 ){
    if ( $d =~ m{$fmt1|$fmt2|$fmt3} ){
        print "day=${d} month=${m} year=${y}\n";
    }
}
```

Si l'une des branches de l'alternative est reconnues alors la table de hachage `%+` contiendra les trois paires clé/valeur recherchées.

8.2.9 Numérotation des groupes dans une alternative

Une autre technique de numérotation (existante elle aussi depuis Perl 5.10) gère le problème du référencement des groupes dans une alternative. Voici une expression rationnelle reconnaissant une heure du jour, à la manière civile ou militaire :

```
if ( $time =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/ ){
    # traitement des heures et minutes
}
```

Le traitement du résultat nécessite une instruction de test supplémentaire pour déterminer si c'est \$1 et \$2 qui contiennent les bonnes valeurs ou au contraire \$3 et \$4. Cela serait plus facile si les deux branches de l'alternative utilisaient toutes les deux les numéros de groupes 1 et 2. C'est exactement ce à quoi sert la construction `(?.*)` autour d'une alternative. Voici une version plus complète de l'expression rationnelle précédente :

```
if ( $time =~ /(?(|(\d\d|\d):(\d\d)|(\d\d)(\d\d))\s+([A-Z][A-Z][A-Z])/ ){
    print "hour=$1 minute=$2 zone=$3\n";
}
```

À l'intérieur de l'alternative, pour chaque branche, la numérotation des groupes commencent à la même position. Après l'alternative, la numérotation continue au numéro juste après le numéro le plus élevé atteint par les branches de l'alternative.

8.2.10 Information de position

En plus de ce qui a été reconnu, depuis la version 5.6.0, Perl fournit aussi la position de ce qui a été reconnu grâce aux tableaux `@-` et `@+`. `$-[0]` est la position du début de l'ensemble de la reconnaissance et `$+[0]` est la position de la fin. De manière similaire, `$-[n]` est la position du début du groupe `$n` et `$+[n]` est la position de sa fin. Si `$n` est indéfini alors `$-[n]` et `$+[n]` le sont aussi. Donc le code :

```
$x = "Mmm...donut, thought Homer";
$x =~ /^(Mmm|Yech)\.\.\.(donut|peas)/; # reconnu
foreach $expr (1..$#-) {
    print "Match $expr: '${$expr}' at position ($-[ $expr ], $+[ $expr ])\n";
}
```

affiche :

```
Match 1: 'Mmm' at position (0,3)
Match 2: 'donut' at position (6,11)
```

Même s'il n'y a aucun regroupement dans l'expression rationnelle, il est encore possible de retrouver exactement ce qui a été reconnu dans la chaîne. Si vous les utilisez, Perl donnera pour valeur à '\$' la partie de la chaîne qui est avant ce qui a été reconnu, à '\$&' la partie de la chaîne qui a été reconnue et à '\$' la partie de la chaîne qui est après ce qui a été reconnu. Un exemple :

```
$x = "the cat caught the mouse";
$x =~ /cat/; # '$' = 'the ', '$&' = 'cat', '$' = ' caught the mouse'
$x =~ /the/; # '$' = '', '$&' = 'the', '$' = ' cat caught the mouse'
```

Lors de la seconde mise en correspondance, '\$' est égal à '' parce que l'expression rationnelle est reconnue au premier caractère dans la chaîne et elle ne voit donc jamais le second 'the'. Il est important de noter que l'utilisation de '\$' et '\$' ralentissent un peu le processus de reconnaissance des expressions rationnelles et que l'utilisation de '\$&' le ralentit aussi, mais un peu moins parce que si ces variables sont utilisées une fois pour une expression rationnelle, elles sont alors calculées pour toutes les expressions rationnelles du programme. Donc si les performances font parties des buts dans votre projet, elles doivent être évitées. Préférez plutôt l'utilisation de '@-' et de '@+' :

```
'$' est la même chose que substr( $x, 0, $-[0] )
'$&' est la même chose que substr( $x, $-[0], $+[0]-$-[0] )
'$' est la même chose que substr( $x, $+[0] )
```

8.2.11 Regroupements sans mémorisation

Un regroupement est indispensable pour construire une alternative mais sa mémorisation n'est pas toujours nécessaire. Une mémorisation inutile peut perturber tant à l'intérieur qu'à l'extérieur de l'expression rationnelle. C'est pour éviter cela qu'on a créé les regroupements sans mémorisation, notés '(?:regexp)', qui permettent toujours de considérer leur contenu comme une seule chose mais sans déclencher sa mémorisation. Les deux types de regroupements peuvent co-exister dans une même expression rationnelle. Puisqu'il n'y a pas extraction de l'information, les regroupements sans mémorisation sont plus rapides que ceux avec. Cela permet aussi de choisir précisément les parties de l'expression rationnelle que l'on veut extraire via les variables de reconnaissance :

```
# reconnaît un nombre, $1-$4 sont affectés,
# mais nous ne voulons que $1
/([+-]?\ *(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?/;

# reconnaît un nombre plus rapidement, seul $1 est affecté
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+)(?:[eE][+-]?\d+)?/;

# reconnaît un nombre, avec $1 = le nombre complet, $2 = l'exposant
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+)(?:[eE]([+-]?\d+)?)/;
```

Les regroupements sans mémorisation sont aussi utiles pour éviter les nuisances que la mémorisation peut avoir sur une opération de découpage (via split) où un regroupement est nécessaire.

```
$x = '12aba34ba5';
@num = split /(a|b)+/, $x; # @num = ('12', 'a', '34', 'b', '5')
@num = split /(?:a|b)+/, $x; # @num = ('12', '34', '5')
```

8.2.12 Reconnaissances répétées

Quelques exemples de la section précédente présentent un défaut. Nous ne reconnaissons que les mots de 3 lettres ou des syllabes de 4 lettres ou moins. Nous aimerions pouvoir reconnaître des mots ou des syllabes de n'importe quelle longueur sans écrire des alternatives horribles comme `\w\w\w\w\w\w\w\w\w\w`.

C'est exactement pour répondre à ce besoin que les meta-caractères *quantificateurs* '?', '*', '+' et '{}' ont été créés. Ils nous permettent de spécifier le nombre de répétitions d'une portion d'une expression rationnelle que nous considérons comme acceptable. Les quantificateurs se placent juste après le caractère, la classe de caractères ou le regroupement dont ils précisent le nombre de répétitions. Ils ont la signification suivante :

- a? signifie : reconnaît 'a' 1 ou 0 fois
- a* signifie : reconnaît 'a' 0 ou plusieurs fois
- a+ signifie : reconnaît 'a' 1 ou plusieurs fois
- a{n,m} signifie : reconnaît au moins n 'a', mais pas plus que m 'a'.
- a{n,} signifie : reconnaît au moins n 'a' ou plus
- a{n} signifie : reconnaît exactement n 'a'

Voici quelques exemples :

```

/[a-z]+\s+\d*/; # reconnaît un mot en minuscules, au moins un espace et
                # n'importe quel nombre de chiffres
/(\w+)\s+\1/;  # reconnaît la répétition d'un mot de n'importe
                # quelle longueur
/y(es)?/i;     # reconnaît 'y', 'Y', ou un 'yes' insensible à la cass
$year =~ /\d{2,4}/; # s'assure que l'année est au moins sur 2 chiffres
                # et pas sur plus de 4 chiffres
$year =~ /\d{4}|\d{2}/; # meilleure reconnaissance ; supprime le cas
                # d'une année sur 3 chiffres
$year =~ /\d{2}(\d{2})?/; # la même chose écrite différemment ; de plus,
                # produit $1 ce que ne faisaient pas les
                # exemples précédents

% simple_grep '^(w+)\1$' /usr/dict/words # c'est simple... non ?
beriberi
booboo
coco
mama
murmur
papa

```

Quel que soit le quantificateur, Perl essaye de reconnaître la chaîne la plus longue possible tant qu'elle peut être mise en correspondance avec l'expression rationnelle. Donc dans /a?..../, Perl essayera d'abord de reconnaître l'expression rationnelle avec un a présent ; en cas d'échec, Perl réessayera sans le a. Avec le quantificateur *, nous obtenons :

```

$x = "the cat in the hat";
$x =~ /^(.*) (cat) (.*)$/; # est reconnue avec
                # $1 = 'the '
                # $2 = 'cat'
                # $3 = ' in the hat'

```

C'est exactement ce que nous attendions, la mise en correspondance trouve le seul cat présent dans la chaîne et se cale dessus. Considérons maintenant cette expression rationnelle :

```

$x =~ /^(.*) (at) (.*)$/; # est reconnue avec
                # $1 = 'the cat in the h'
                # $2 = 'at'
                # $3 = '' (reconnaissance de la chaîne vide)

```

On aurait pu croire que Perl trouverait le at dans cat et se serait arrêté là mais cela n'aurait pas donné la chaîne la plus longue possible pour le premier quantificateur *. En fait, le premier quantificateur * consomme la plus grande partie possible de la chaîne tout en laissant à l'expression rationnelle la possibilité d'être reconnue. Dans cet exemple, cela signifie que la séquence at est mise en correspondance avec le dernier cat de la chaîne. L'autre principe important illustré ici est que lorsque qu'il y a plusieurs éléments dans une expression rationnelle, c'est le quantificateur le plus à gauche, s'il existe, qui est servi en premier et qui laisse le minimum au reste de l'expression rationnelle. Donc dans notre exemple, c'est le premier quantificateur * qui consomme la plus grande partie de la chaîne alors que le second quantificateur * obtient la chaîne vide. Les quantificateurs qui consomment autant que possible sont qualifiés de *maximaux* ou de *gourmands*.

Lorsque une expression rationnelle peut être mise en correspondance avec une chaîne de différentes façons, nous pouvons utiliser les principes suivants pour prédire la manière dont elle sera reconnue :

- Principe 0 : Tout d'abord, une expression rationnelle sera toujours reconnue à la position la plus à gauche possible dans la chaîne.

- Principe 1 : Dans un choix $a|b|c\dots$, c'est la branche la plus à gauche permettant une reconnaissance de l'ensemble de l'expression rationnelle qui sera choisie en premier.
- Principe 2 : Les quantificateurs gourmands comme $?$, $*$, $+$ et $\{n,m\}$ consomme la plus grande partie possible de la chaîne tant qu'ils permettent encore de reconnaître l'ensemble de l'expression rationnelle.
- Principe 3 : s'il existe plusieurs éléments dans une expression rationnelle, le quantificateur gourmand le plus à gauche, s'il existe, sera mis en correspondance avec la partie de la chaîne la plus longue possible tout en gardant possible la reconnaissance de toute l'expression rationnelle. Le quantificateur gourmand suivant, s'il existe, sera mis en correspondance avec la partie la plus longue possible de ce qui reste de la chaîne tout en gardant possible la reconnaissance de toute l'expression rationnelle. Et ainsi de suite, jusqu'à la reconnaissance complète de l'expression rationnelle.

Comme vu précédemment, le Principe 0 est prioritaire sur tous les autres - l'expression rationnelle est reconnue le plus tôt possible en utilisant les autres principes pour déterminer comment l'expression rationnelle est reconnue à cette position la plus à gauche.

Voici des exemples qui montrent l'application de ces principes :

```
$x = "The programming republic of Perl";
$x =~ /^(.+)(e|r)(.*)$/; # est reconnue avec
                        # $1 = 'The programming republic of Pe'
                        # $2 = 'r'
                        # $3 = 'l'
```

L'expression rationnelle est reconnue à la position la plus tôt, 'T'. On pourrait penser que le e le plus à gauche de l'alternative devrait être reconnu mais le r produit une chaîne plus longue pour le premier quantificateur.

```
$x =~ /(m{1,2})(.*)$/; # est reconnue avec
                        # $1 = 'mm'
                        # $2 = 'ing republic of Perl'
```

Ici, la mise en correspondance au plus tôt se fait au premier 'm' de programming. $m\{1,2\}$ est le premier quantificateur et donc il cherche la correspondance maximale mm.

```
$x =~ /.*(m{1,2})(.*)$/; # est reconnue avec
                        # $1 = 'm'
                        # $2 = 'ing republic of Perl'
```

Ici, l'expression rationnelle est mise en correspondance dès le début de la chaîne. Le premier quantificateur $.*$ consomme le plus de caractères possibles en ne laissant qu'un seul 'm' pour le second quantificateur $m\{1,2\}$.

```
$x =~ /(.(?)(m{1,2})(.*)$/; # est reconnue avec
                        # $1 = 'a'
                        # $2 = 'mm'
                        # $3 = 'ing republic of Perl'
```

Ici, $.?$ consomme le maximum de caractères (un caractère) à la position la plus à gauche possible dans la chaîne, le 'a' dans programming, en laissant l'opportunité à $m\{1,2\}$ de correspondre aux deux m. Finalement :

```
"aXXXb" =~ /(X*)/; # est reconnue avec $1 = ''
```

parce qu'il peut reconnaître zéro 'X' au tout début de la chaîne. Si vous voulez réellement reconnaître au moins un 'X', utilisez $X+$ au lieu de X^* .

Parfois la gourmandise n'est pas une bonne chose. Dans ce cas, nous aimerions des quantificateurs qui reconnaissent une partie *minimale* de la chaîne plutôt que *maximale*. Pour cela, Larry Wall a créé les quantificateurs *minimaux* ou quantificateurs *sobres* : $??$, $*?$, $+$? et $\{n\}?$. Ce sont les quantificateurs habituels auxquels on ajoute un ?. Ils ont la sémantique suivante :

- $a??$ signifie : reconnaît un 'a' 0 ou 1 fois. Essaie 0 d'abord puis 1 ensuite.
- $a*?$ signifie : reconnaît zéro 'a' ou plus mais un minimum de fois.
- $a+?$ signifie : reconnaît un 'a' ou plus mais un minimum de fois.
- $a\{n,m\}?$ signifie : reconnaît entre n et m 'a' mais un minimum de fois.
- $a\{n,\}?$ signifie : reconnaît au moins n 'a' mais un minimum de fois.
- $a\{n\}?$ = reconnaît exactement n 'a'. C'est équivalent à $a\{n\}$. Ce n'est donc que pour rendre la notation cohérente.

Reprenons les exemples précédents mais avec des quantificateurs minimaux :

```
$x = "The programming republic of Perl";
$x =~ /^(.+?)(e|r)(.*)$/; # est reconnue avec
# $1 = 'Th'
# $2 = 'e'
# $3 = ' programming republic of Perl'
```

La chaîne minimale permettant à la fois de reconnaître le début de chaîne `^` et l'alternative est `Th` avec l'alternative `e|r` qui reconnaît `e`. Le second quantificateur est libre de consommer tout le reste de la chaîne.

```
$x =~ /(m{1,2}?)(.*)$/; # est reconnue avec
# $1 = 'm'
# $2 = 'ming republic of Perl'
```

La première position à partir de laquelle cette expression rationnelle peut être reconnue et le premier `'m'` de `programming`. À cette position, le sobre `m{1,2}?` reconnaît juste un `'m'`. Ensuite, bien que le second quantificateur `.*` préfère reconnaître un minimum de caractères, il est contraint par l'ancre de fin de chaîne `$` à reconnaître tout le reste de la chaîne.

```
$x =~ /(.*?) (m{1,2}?) (.*)$/; # est reconnue avec
# $1 = 'The progra'
# $2 = 'm'
# $3 = 'ming republic of Perl'
```

Dans cette expression rationnelle, vous espérez peut-être que le premier quantificateur minimal `.*` soit mis en correspondance avec la chaîne vide puisqu'il n'est pas contraint à s'ancrer en début de chaîne par `^`. Mais ici le principe 0 s'applique. Puisqu'il est possible de reconnaître l'ensemble de l'expression rationnelle en s'ancrant au début de la chaîne, ce sera cet ancrage qui sera choisi. Donc le premier quantificateur doit reconnaître tout jusqu'au premier `m` et le troisième quantificateur reconnaît le reste de la chaîne.

```
$x =~ /(.*?) (m{1,2}) (.*)$/; # est reconnue avec
# $1 = 'a'
# $2 = 'mm'
# $3 = 'ing republic of Perl'
```

Comme dans l'expression rationnelle précédente, le premier quantificateur peut correspondre au plus tôt sur le `'a'`, c'est donc ce qui est retenu. Le second quantificateur est gourmand donc il reconnaît `mm` et le troisième reconnaît le reste de la chaîne.

Nous pouvons modifier le principe 3 précédent pour prendre en compte les quantificateurs sobres :

- Principe 3: s'il existe plusieurs éléments dans une expression rationnelle, le quantificateur gourmand (ou sobre) le plus à gauche, s'il existe, sera mis en correspondance avec la partie de la chaîne la plus longue (ou la plus courte) possible tout en gardant possible la reconnaissance de toute l'expression rationnelle. Le quantificateur gourmand (ou sobre) suivant, s'il existe, sera mis en correspondance avec la partie la plus longue (ou la plus courte) possible de ce qui reste de la chaîne tout en gardant possible la reconnaissance de toute l'expression rationnelle. Et ainsi de suite, jusqu'à la reconnaissance complète de l'expression rationnelle.

Comme avec les alternatives, les quantificateurs sont susceptibles de déclencher des retours arrière. Voici l'analyse pas à pas d'un exemple :

```
$x = "the cat in the hat";
$x =~ /^(.*) (at) (.*)$/; # est reconnue avec
# $1 = 'the cat in the h'
# $2 = 'at'
# $3 = '' (reconnaissance de la chaîne vide)
```

1. On démarre avec la première lettre de la chaîne `'t'`.
2. Le premier quantificateur `'.*'` commence par reconnaître l'ensemble de la chaîne `'the cat in the hat'`.
3. Le `'a'` de l'élément `'at'` ne peut pas être mis en correspondance avec la fin de la chaîne. Retour arrière d'un caractère.

4. Le 'a' de l'élément 'at' ne peut pas être mis en correspondance avec la dernière lettre de la chaîne 't'. Donc à nouveau un retour arrière d'un caractère.
5. Maintenant nous pouvons reconnaître le 'a' et le 't'.
6. On continue avec le troisième élément '.*'. Puisque nous sommes à la fin de la chaîne, '.*' ne peut donc reconnaître que 0 caractère. On lui affecte donc la chaîne vide.
7. C'est fini.

La plupart du temps, tous ces aller-retours ont lieu très rapidement et la recherche est rapide. Par contre, il existe quelques expressions rationnelles pathologiques dont le temps d'exécution augmente exponentiellement avec la taille de la chaîne. Une structure typique est de la forme :

```
/(a|b+)*;/
```

Le problème est l'imbrication de deux quantificateurs indéterminés. Il y a de nombreuses manières de partitionner une chaîne de longueur n entre le $+$ et le $*$: une répétition avec $b+$ de longueur n , deux répétitions avec le premier $b+$ de longueur k et le second de longueur $n-k$, m répétitions dont la somme des longueurs est égale à n , etc. En fait, le nombre de manières différentes de partitionner une chaîne est exponentielle en fonction de sa longueur. Une expression rationnelle peut être chanceuse et être reconnue très tôt mais s'il n'y a pas de reconnaissance possible, Perl essayera *toutes* les possibilités avant de conclure à l'échec. Donc, soyez prudents lorsque vous imbriquez des $*$, des $\{n, m\}$ et/ou des $+$. Le livre *Mastering Regular Expressions* par Jeffrey Friedl donne de très bonnes explications sur ce problème en particulier et sur tous les autres problèmes de performances.

8.2.13 Quantificateurs possessifs

Effectuer des retours arrières pendant la recherche acharnée d'une reconnaissance peut être une perte de temps, particulièrement si cette recherche est sûre d'échouer. Considérons cette simple expression :

```
/^\w+\s+\w+$/; # un mot, un espace, un mot
```

À chaque fois qu'elle est appliquée à une chaîne qui ne correspond pas à ce qui est attendu telle "abc " ou "abc def ", le moteur de regexp effectue des retours arrières, approximativement pour chaque caractère de la chaîne. Or nous savons qu'il n'y a pas d'autre choix que de consommer *tous* les caractères mot initiaux pour pouvoir reconnaître la première répétition, que *tous* les espaces qui suivent doivent être mangés par la seconde répétition et il en est de même pour le deuxième mot.

Avec l'introduction des *quantificateurs possessifs* en Perl 5.10, nous avons moyen d'empêcher les retours arrières du moteur de regexp, en ajoutant un $+$ après un quantificateur usuel. Cela les rend à la fois gourmands et radins ; une fois qu'ils sont reconnus, ils ne rendent plus aucun caractère pour essayer une autre solution. Ils ont la signification suivante :

- $a\{n, m\}+$ signifie : reconnaît au moins n fois, mais pas plus que m fois, autant de fois que possible et n'essaye pas d'autre possibilité. $a?+$ est un raccourci pour $a\{0, 1\}+$.
- $a\{n, \}+$ signifie : reconnaît au moins n fois et autant de fois que possible et n'essaye pas d'autre possibilité. $a*+$ est un raccourci pour $a\{0, \}+$ et $a++$ est un raccourci pour $a\{1, \}+$.
- $a\{n\}+$ signifie : reconnaît exactement n fois. Il n'existe que pour rendre la notation cohérente.

Ces quantificateurs possessifs ne sont que des cas spéciaux du concept plus général de *sous-expressions indépendantes*, voir ci-dessous.

Comme exemple d'utilisation des quantificateurs possessifs, nous considérons la reconnaissance des chaînes de caractères entre guillemets, telles qu'elles apparaissent dans de nombreux langages. Le backslash est utilisé comme caractère d'échappement pour indiquer que le caractère suivant doit être considéré littéralement, comme n'importe quel autre caractère de la chaîne. Donc, après le guillemet ouvrant, nous attendons une séquence (éventuellement vide) de suites de caractères composées soit de plusieurs caractères différents du guillemet fermant et du backslash, soit d'un caractère précédé du caractère d'échappement (le backslash).

```
/"(?: [^\\" ]++ |\\.)*+"/;
```

8.2.14 Construction d'une expression rationnelle

À ce point, nous avons couvert tous les concepts de base des expressions rationnelles. Nous pouvons donc présenter un exemple plus complet d'utilisation des expressions rationnelles. Nous allons construire une expression rationnelle qui reconnaît les nombres.

La première tâche de la construction d'une expression rationnelle consiste à décider ce que nous voulons reconnaître et ce que nous voulons exclure. Dans notre cas, nous voulons reconnaître à la fois les entiers et les nombres en virgule flottante et nous voulons rejeter les chaînes qui ne sont pas des nombres.

La tâche suivante permet de découper le problème en plusieurs petits problèmes qui seront plus simplement transformés en expressions rationnelles.

Le cas le plus simple concerne les entiers. Ce sont une suite de chiffres précédée d'un éventuel signe. Les chiffres peuvent être représentés par `\d+` et le signe peut être reconnu par `[+-]`. Donc l'expression rationnelle pour les entiers est :

```
/[+-]?\d+;/ # reconnaît les entiers
```

Un nombre en virgule flottante contient potentiellement un signe, une partie entière, un séparateur décimal (un point), une partie décimale et un exposant. Plusieurs de ces parties sont optionnelles donc nous devons vérifier les différentes possibilités. Les nombres en virgule flottante bien formés contiennent entre autres 123., 0.345, .34, -1e6 et 25.4E-72. Comme pour les entiers, le signe au départ est complètement optionnel et peut être reconnu par `[+-]?`. Nous pouvons voir que s'il n'a pas d'exposant, un nombre en virgule flottante doit contenir un séparateur décimal ou alors c'est un entier. Nous pourrions être tentés d'utiliser `\d*\.\d*` mais cela pourrait aussi reconnaître un séparateur décimal seul (qui n'est pas un nombre). Donc, les trois cas de nombres sans exposant sont :

```
/[+-]?\d+\./; # 1., 321., etc.
/[+-]?\.\d+;/ # .1, .234, etc.
/[+-]?\d+\.\d+;/ # 1.0, 30.56, etc.
```

On peut combiner cela en une seule expression rationnelle :

```
/[+-]?(\d+\.\d+|\d+\.\.\d+)/; # virgule flottante, sans exposant
```

Dans ce choix, il est important de placer '`\d+\.\d+`' avant '`\d+\.\.`'. Si '`\d+\.`' était en premier, l'expression rationnelle pourrait être reconnue en ignorant la partie décimale du nombre.

Considérons maintenant les nombres en virgule flottante avec exposant. Le point clé ici est que les nombres avec séparateur décimal *ainsi* que les entiers sont autorisés devant un exposant. Ensuite la reconnaissance de l'exposant, comme celle du signe, est indépendante du fait que le nombre possède ou non une partie décimale. Elle peut donc être découpée de la mantisse. La forme de l'expression rationnelle complète devient donc claire maintenant :

```
/^(signe optionnel)(entier | mantisse)(exposant optionnel)$/;
```

L'expression est un `e` ou un `E` suivi d'un entier. Donc l'expression rationnelle de l'exposant est :

```
/[eE][+-]?\d+;/ # exposant
```

En assemblant toutes les parties ensemble nous obtenons l'expression rationnelle qui reconnaît les nombres :

```
/^[+-]?(\d+\.\d+|\d+\.\.\d+)([eE][+-]?\d+)?$/; # Ta da!
```

De longues expressions rationnelles comme celle-ci peuvent peut-être impressionner vos amis mais elles sont difficiles à déchiffrer. Dans des situations complexes comme celle-ci, le modificateur `//x` est très pratique. Il vous permet de placer des espaces et des commentaires n'importe où dans votre expression sans en changer la signification. En l'utilisant, nous pouvons réécrire notre expression sous une forme plus plaisante :

```
/^
  [+-]?          # en premier, reconnaissance du signe optionnel
  (
    \d+\.\d+    # mantisse de la forme a.b
    |\d+\.\     # mantisse de la forme a.
    |\.\d+      # mantisse de la forme .b
    |\d+        # entier de la forme a
  )
  ([eE][+-]?\d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;
```

Si les espaces ne sont pas pris en compte, comment inclure un caractère espace dans une telle expression rationnelle ? Il suffit de le «backslasher» '\ ' ou de le placer dans une classe de caractères []. La même chose est valable pour le dièse : utilisez # ou [#]. Par exemple, Perl autorise un espace entre le signe et la mantisse ou l'entier. Nous pouvons ajouter cela dans notre expression rationnelle comme suit :

```

/^
  [+]? \ *      # en premier, reconnaissance du signe optionnel *et des espaces*
  (
    # ensuite reconnaissance d'un entier ou d'un
    # nombre en virgule flottante
    \d+ \. \d+  # mantisse de la forme a.b
    | \d+ \.    # mantisse de la forme a.
    | \. \d+    # mantisse de la forme .b
    | \d+      # entier de la forme a
  )
  ([eE][+-]? \d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;

```

Sous cette forme, il est plus simple de découvrir un moyen de simplifier le choix. Les branches 1, 2 et 4 du choix commencent toutes par \d+ qu'on doit donc pouvoir factoriser :

```

/^
  [+]? \ *      # en premier, reconnaissance du signe optionnel
  (
    # ensuite reconnaissance d'un entier ou d'un
    # nombre en virgule flottante
    \d+         # on commence par a ...
    (
      \. \d*    # mantisse de la forme a. ou a.b
    )?         # ? pour les entiers de la forme a
    | \. \d+    # mantisse de la forme .b
  )
  ([eE][+-]? \d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;

```

ou écrit dans sa forme compacte :

```

/^ [+]? \ * ( \d+ ( \. \d* )? | \. \d+ ) ([eE][+-]? \d+ )? $/;

```

C'est notre expression rationnelle finale. Pour récapituler, nous avons construit notre expression rationnelle :

- en spécifiant la tâche en détail,
- en découpant le problème en plus petites parties,
- en transformant les petites parties en expressions rationnelles,
- en combinant les expressions rationnelles,
- et en optimisant l'expression rationnelle combinée finale.

On peut faire le parallèle avec les différentes étapes de l'écriture d'un programme. C'est normal puisque les expressions rationnelles sont des programmes écrits dans un petit langage informatique de spécification de motifs.

8.2.15 Utilisation des expressions rationnelles en Perl

Pour terminer cette première partie, nous allons examiner brièvement comment les expressions rationnelles sont utilisées dans un programme Perl. Comment s'intègrent-elles à la syntaxe Perl ?

Nous avons déjà parlé de l'opérateur de recherche de correspondances dans sa forme par défaut /*regex*/ ou avec des délimiteurs arbitraires *m! regex!*. Nous avons utilisé l'opérateur de mise en correspondance =~ ainsi que sa négation !~ pour rechercher les correspondances. Associé avec l'opérateur de recherche de correspondances, nous avons présenté les modificateurs permettant de traiter de simples lignes //s, des multi-lignes //m, des expressions insensibles à la casse //i et des expressions étendues //x. Il y a encore quelques points que vous devez connaître à propos des opérateurs de mise en correspondance.

Optimiser l'évaluation des expressions rationnelles

Nous avons montré que les variables étaient substituées avant l'évaluation de l'expression rationnelle :

```
$pattern = 'Seuss';
while (<>) {
    print if /$pattern/;
}
```

Cela affichera toutes les lignes contenant le mot Seuss. Par contre, ce n'est pas aussi efficace qu'il y paraît car Perl doit réévaluer (ou compiler) `$pattern` à chaque passage dans la boucle. Si `$pattern` ne doit pas changer durant la vie du script, nous pouvons ajouter le modificateur `//o` qui demande à Perl de n'effectuer la substitution de variables qu'une seule fois :

```
#!/usr/bin/perl
#   grep simple amélioré
$regexp = shift;
while (<>) {
    print if /$regexp/o; # cela va plus vite...
}
```

Interdire les substitutions

Si vous changez `$pattern` après la première substitution, Perl l'ignorera. Si vous voulez que perl n'effectue aucune substitution, utilisez le délimiteur spécial `m` :

```
@pattern = ('Seuss');
while (<>) {
    print if m'@pattern'; # correspond littéralement avec '@pattern',
                          # pas avec 'Seuss'
}
```

Comme pour les chaînes, `m` agit exactement comme les apostrophes mais sur les expressions rationnelles ; tout autre délimiteur agit comme des guillemets. Si l'expression rationnelle s'évalue à la chaîne vide, la dernière expression rationnelle utilisée avec succès sera utilisée à la place. Donc :

```
"dog" =~ /d/; # 'd' correspond
"dogbert" =~ //; # mise en correspondance par
                 # l'expression rationnelle 'd' précédente
```

Reconnaissance globale

Le deux modificateurs restant (`//g` et `//c`) concernent les recherches multiples. Le modificateur `//g` signifie recherche globale et autorise l'opérateur de mise en correspondance à correspondre autant de fois que possible. Dans un contexte scalaire, des invocations successives d'une expression rationnelle modifiée par `//g` appliquée à une même chaîne passeront d'une correspondance à une autre, en se souvenant de la position atteinte dans la chaîne explorée. Vous pouvez consulter ou modifier cette position grâce à la fonction `pos()`.

L'utilisation de `//g` est montrée dans l'exemple suivant. Supposons une chaîne constituée de mots séparés par des espaces. Si nous connaissons à l'avance le nombre de mots, nous pouvons extraire les mots en utilisant les regroupements :

```
$x = "cat dog house"; # 3 mots
$x =~ /^s*(\w+)\s+(\w+)\s+(\w+)\s*$/; # correspond avec
                                         # $1 = 'cat'
                                         # $2 = 'dog'
                                         # $3 = 'house'
```

Mais comment faire si le nombre de mots est indéterminé ? C'est pour ce genre de tâche qu'on a créé `//g`. Pour extraire tous les mots, on utilise l'expression rationnelle simple `(\w+)` et on boucle sur toutes les correspondances possibles grâce à `/(\w+)/g` :

```
while ($x =~ /(\w+)/g) {
    print "Le mot $1 se termine à la position ", pos $x, "\n";
}
```

qui affiche

```
Le mot cat se termine à la position 3
Le mot dog se termine à la position 7
Le mot house se termine à la position 13
```

Un échec de la mise en correspondance ou un changement de chaîne cible réinitialise la position. Si vous ne voulez pas que la position soit réinitialisée après un échec, ajoutez le modificateur `//c` comme dans `/regexp/gc`. La position courante dans la chaîne est associée à la chaîne elle-même et non à l'expression rationnelle. Cela signifie que des chaînes différentes ont des positions différentes et que ces positions peuvent être modifiées indépendamment.

Dans un contexte de liste, `//g` retourne la liste de tous les groupes mis en correspondance ou, s'il n'y a pas de groupes, la liste de toutes les reconnaissances de l'expression rationnelle entière. Donc si nous ne voulons que les mots, nous pouvons faire :

```
@words = ($x =~ /(\w+)/g); # correspond avec
                        # $word[0] = 'cat'
                        # $word[1] = 'dog'
                        # $word[2] = 'house'
```

Étroitement associé avec le modificateur `//g`, il existe l'ancre `\G`. L'ancre `\G` est mis en correspondance avec le point atteint lors d'une reconnaissance précédente par `//g`. `\G` permet de faire de la reconnaissance dépendante du contexte :

```
$metric = 1; # utilisation des unités métriques
...
$x = <FILE>; # lecture des mesures
$x =~ /^( [+ ]? \d+ ) \s* /g; # obtention de la valeur
$weight = $1;
if ($metric) { # vérification d'erreur
    print "Units error!" unless $x =~ /\Gkg\./g;
}
else {
    print "Units error!" unless $x =~ /\Glbs\./g;
}
$x =~ /\G\s+(widget|sprocket)/g; # suite du traitement
```

La combinaison de `//g` et de `\G` permet le traitement pas à pas d'une chaîne et l'utilisation de Perl pour déterminer la suite du traitement. Actuellement, l'ancre `\G` n'est réellement utilisable que si elle est utilisée en début de motif.

`\G` est aussi pratique lors du traitement par des expressions rationnelles d'enregistrement à taille fixe. Supposons une séquence d'ADN, encodée comme une suite de lettres ATCGTTGAAT... et que vous voulez trouver tous les codons du type TGA. Dans une séquence, les codons sont des séquences de 3 lettres. Nous pouvons donc considérer une chaîne d'ADN comme une séquence d'enregistrements de 3 lettres. L'expression rationnelle naïve :

```
# C'est "ATC GTT GAA TGC AAA TGA CAT GAC"
$dna = "ATCGTTGAATGCAAATGACATGAC";
$dna =~ /TGA/;
```

ne marche pas ; elle retrouvera un TGA mais il n'y aura aucune garantie que cette correspondance soit alignée avec une limite de codon, e.g., la sous-chaîne GTT GAA donnera une correspondance. Une meilleure solution est :

```
while ($dna =~ /(\w\w\w)*?TGA/g) { # remarquez le minimal *?
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

qui affichera :

```
Got a TGA stop codon at position 18
Got a TGA stop codon at position 23
```

La position 18 est correcte mais pas la position 23. Que s'est-il passé ?

Notre expression rationnelle fonctionne bien tant qu'il reste de bonnes correspondances. Puis l'expression rationnelle ne trouve plus de TGA bien synchronisé et réessaie après s'être décalée d'un caractère à chaque fois. Ce n'est pas ce que nous voulons. La solution est l'utilisation de `\G` pour ancrer notre expression rationnelle sur une limite de codon :

```
while ($dna =~ /\G(\w\w\w)*?TGA/g) {
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

qui affiche :

```
Got a TGA stop codon at position 18
```

qui est la bonne réponse. Cet exemple illustre qu'il ne suffit pas de reconnaître ce que l'on cherche, il faut aussi rejeter ce qu'on ne veut pas.

Recherche et remplacement

Les expressions rationnelles jouent aussi un grand rôle dans les opérations de *recherche et remplacement* en Perl. Une recherche/remplacement est accomplie via l'opérateur `s///`. La forme générale est `s/regexp/remplacement/modificateurs` avec l'application de tout ce que nous connaissons déjà sur les expressions rationnelles et les modificateurs. La partie `remplacement` est comme une chaîne entre guillemets de Perl qui remplacera la partie de la chaîne reconnue par l'expression rationnelle `regexp`. L'opérateur `=~` est aussi utilisé pour associer une chaîne avec `s///`. Si on veut l'appliquer à `$_`, on peut omettre `$_ =~`. S'il y a correspondance, `s///` renvoie le nombre de substitutions effectuées sinon il retourne faux. Voici quelques exemples :

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contient "Time to feed the hacker!"
if ($x =~ s/(Time.*hacker)!$/!$1 now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^(.*)'$/!$1/; # suppression des apostrophes,
# $y contient "quoted words"
```

Dans le dernier exemple, la chaîne entière est reconnue mais seule la partie à l'intérieur des apostrophes est dans un groupe. Avec l'opérateur `s///`, les variables `$1`, `$2`, etc. sont immédiatement disponibles pour une utilisation dans l'expression de remplacement. Donc, nous utilisons `$1` pour remplacer la chaîne entre apostrophes par ce qu'elle contient. Avec le modificateur global, `s///g` cherchera et remplacera toutes les occurrences de l'expression rationnelle dans la chaîne :

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # ne fait pas tout :
# $x contient "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g; # fait tout :
# $x contient "I batted four for four"
```

Si vous préférez `'regex'` à la place de `'regexp'` dans ce tutoriel, vous pourriez utiliser le programme suivant pour les remplacer :

```
% cat > simple_replace
#!/usr/bin/perl
$regexp = shift;
$replacement = shift;
while (<>) {
    s/$regexp/$replacement/go;
    print;
}
^D
```

```
% simple_replace regexp regex perlretut.pod
```

Dans `simple_replace` nous utilisons le modificateur `s///g` pour remplacer toutes les occurrences de l'expression rationnelle à chaque ligne et le modificateur `s///o` pour compiler l'expression rationnelle une seule fois pour toutes. Comme avec `simple_grep`, les deux instructions `print` et `s/$regexp/$remplacement/go` utilisent implicitement la variable `$_`.

Un modificateur spécifique à l'opération de recherche/remplacement est le modificateur d'évaluation `s///e`. `s///e` ajoute un `eval{...}` autour de la chaîne de remplacement et le résultat de cette évaluation est substitué à la sous-chaîne mise en correspondance. `s///e` est utile si vous avez besoin de faire un traitement sur le texte à remplacer. Cet exemple compte la fréquence des lettres dans une ligne :

```
$x = "Bill the cat";
$x =~ s/(.)/$chars{$1}++;$1/eg; # Le $1 final remplace le caractère par lui-même
print "frequency of '$_' is $chars{$_}\n"
      foreach (sort {$chars{$b} <=> $chars{$a}} keys %chars);
```

qui affiche :

```
frequency of ' ' is 2
frequency of 't' is 2
frequency of 'l' is 2
frequency of 'B' is 1
frequency of 'c' is 1
frequency of 'e' is 1
frequency of 'h' is 1
frequency of 'i' is 1
frequency of 'a' is 1
```

Comme pour l'opérateur `m//`, `s///` peut utiliser d'autres délimiteurs comme `s!!!` ou `s{ }{ }` et même `s{ }{ }/`. Si les délimiteurs sont des apostrophes `s''` alors l'expression rationnelle et le remplacement sont traités comme des chaînes entre apostrophes et aucune interpolation de variables n'est effectuée. `s///` dans un contexte de liste retourne la même chose qu'en contexte scalaire, c'est à dire le nombre de substitutions effectuées.

La fonction de découpage (split)

La fonction `split()` peut, elle aussi, utiliser un opérateur de mise en correspondance `m//` pour découper une chaîne. `split /regexp/, chaine, limite` découpe la chaîne en une liste de sous-chaînes et retourne cette liste. L'expression rationnelle `regexp` est utilisée pour reconnaître la séquence de caractères qui servira de séparateur lors du découpage de chaîne. La `limite`, si elle est présente, indique le nombre maximal de morceaux lors du découpage. Par exemple, pour découper une chaîne en mots, utilisez :

```
$x = "Calvin and Hobbes";
@words = split /\s+/, $x; # $word[0] = 'Calvin'
                        # $word[1] = 'and'
                        # $word[2] = 'Hobbes'
```

Si l'expression rationnelle vide `//` est utilisée alors l'expression rationnelle correspond partout et la chaîne est découpée en caractères individuels. Si l'expression rationnelle contient des groupes alors la liste résultante contient aussi les sous-chaînes reconnues par les groupes. Par exemple :

```
$x = "/usr/bin/perl";
@dirs = split m!/!, $x; # $dirs[0] = ''
                        # $dirs[1] = 'usr'
                        # $dirs[2] = 'bin'
                        # $dirs[3] = 'perl'
@parts = split m!(/)! , $x; # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
                            # $parts[5] = '/'
                            # $parts[6] = 'perl'
```

Puisque le premier caractère de `$X` est reconnu par l'expression rationnelle, `split` ajoute un élément initial vide à la liste. Si vous avez tout lu jusqu'ici, félicitations ! Vous possédez maintenant tous les outils de base pour utiliser les expressions rationnelles afin de résoudre de nombreux problèmes de traitement de textes. Si c'est la première fois que vous lisez ce tutoriel, vous devriez vous arrêter ici et jouer quelques temps avec les expressions rationnelles... La Partie 2 concerne des aspects plus ésotériques des expressions rationnelles et ces concepts ne sont certainement pas nécessaires au début.

8.3 Partie 2: au-delà

Bon, vous connaissez les bases des expressions rationnelles et vous voulez en savoir plus. Si la mise en correspondance d'une expression rationnelle est analogue à une marche en forêt alors les outils dont nous avons parlé dans la partie 1 sont la carte et le compas, des outils basiques que nous utilisons tout le temps. La plupart des outils de la partie 2 sont alors analogues à un lance fusées éclairantes ou à un téléphone satellite. On ne les utilise pas très souvent mais, en cas de besoin, ils sont irremplaçables.

Ce qui suit présente les fonctionnalités les plus avancées, les moins utilisées ou les plus ésotériques des expressions rationnelles de Perl. Dans cette seconde partie, nous supposons que vous êtes à l'aise avec les outils de base pour nous concentrer sur ces nouvelles fonctionnalités.

8.3.1 Les plus des caractères, des chaînes et des classes de caractères

Il y a de nombreuses séquences d'échappement et classes de caractères dont nous n'avons pas encore parlées.

Il existe plusieurs séquences d'échappement qui convertissent les caractères ou les chaînes entre majuscules et minuscules, et elles peuvent être utilisées dans les expressions rationnelles. `\l` et `\u` convertissent le caractère suivant respectivement en minuscule ou en majuscule :

```
$x = "perl";
$string =~ /\u$x/; # reconnaît 'Perl' dans $string
$x = "M(rs?|s)\."; # notez le double backslash
$string =~ /\l$x/; # reconnaît 'mr.', 'mrs.' et 'ms.',
```

`\L` et `\U` convertissent une sous-chaîne entière délimitée `\E`, ou stoppée par un autre `\L` ou `\U`.

```
$x = "This word is in lower case:\L SHOUT\E";
$x =~ /shout/; # correspond
$x = "I STILL KEYPUNCH CARDS FOR MY 360"
$x =~ /\Ukeypunch/; # correspond
```

S'il n'y a pas de `\E`, le changement de casse a lieu jusqu'à la fin de la chaîne. Les expressions rationnelles `\L\u$word` ou `\U\L$word` convertissent le premier caractère de `$word` en majuscule et les autres caractères en minuscules.

Les caractères de contrôle peuvent être codés via `\c`. Le caractère control-z sera mis en correspondance avec `\cZ`. La séquence d'échappement `\Q...\E` protège la plupart des caractères non-alphabétiques. Par exemple :

```
$x = "\QThat !^*&%~& cat!";
$x =~ /\Q!^*&%~&\E/; # correspond
```

Les caractères `$` et `@` ne sont pas protégés donc les variables peuvent encore être interpolées.

Avec Perl 5.6.0, les expressions rationnelles peuvent gérer plus que le jeu de caractères ASCII. Perl gère maintenant *Unicode*, un standard pour représenter les alphabets de la plupart des langues écrites complétés de nombreux symboles. Les chaînes Perl sont des chaînes Unicode et, de fait, elles peuvent contenir des caractères dont la valeur (le point de code ou numéro de caractère) est supérieure à 255.

Quelles conséquences sur les expressions rationnelles ? Les utilisateurs d'expressions rationnelles n'ont pas besoin de connaître la représentation interne des chaînes de perl. Par contre, il faut qu'ils sachent 1) comment représenter les caractères Unicode dans une expression rationnelle et 2) que toutes les opérations de mise en correspondance traiteront une chaîne comme une séquence de caractères Unicode et non comme une séquence d'octets. La réponse à la question 1) est que les caractères Unicode plus grands que `chr(127)` peuvent être représentés en utilisant la notation `\x{hex}`, sachant que les notations `\0` octale et `\x` hexadécimale (sans les guillemets) ne peuvent pas dépasser 255.

```
/\x{263a}/; # correspond à l'émoticon souriant d'Unicode :)
```


NOTE : en Perl 5.6.0 il fallait ajouter la directive `use utf8` pour activer les fonctionnalités Unicode. Ce n'est plus nécessaire : la quasi totalité des fonctionnalités Unicode ne nécessitent plus la directive ou pragma `utf8`. (Le seul cas où cela reste nécessaire est celui où votre script Perl lui-même est écrit en Unicode encodé en UTF-8.)

Se souvenir de la séquence hexadécimale d'un caractère Unicode ou décoder les séquences hexadécimales d'un autre dans une expression rationnelle est presque aussi fun que de coder en langage machine. Un autre moyen de spécifier des caractères Unicode est d'utiliser des *caractères nommés* via la séquence d'échappement `\N{nom}`. `nom` est le nom d'un caractère Unicode comme spécifié dans le standard Unicode. Par exemple, si vous voulez représenter ou reconnaître le signe astrologique de la planète Mercure, vous pourriez utiliser :

```
use charnames ":full"; # utilise les caractères nommés
                        # avec les noms Unicode complet
$x = "abc\N{MERCURY}def";
$x =~ /\N{MERCURY}/; # correspond
```

Il est aussi possible d'utiliser les noms courts ou restreints à certains alphabets :

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ":short";
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(greek);
print "\N{sigma} is Greek sigma\n";
```

Une liste de tous les noms complets est disponible dans le fichier `Names.txt` du répertoire `lib/perl5/X.X.X/unicode` (où `X.X.X` est le numéro de la version de perl installée sur votre système).

La réponse au point 2), depuis la version 5.6.0, est qu'une expression rationnelle utilise les caractères Unicode. En interne, selon son histoire, une chaîne peut être codée en UTF-8 ou en codage natif sur 8 bits, mais conceptuellement cela reste une séquence de caractères et non d'octets. Voir *perlunitut* pour un tutoriel à ce sujet.

Voyons maintenant les classes de caractères Unicode. Comme pour les caractères Unicode, il existe des noms pour les classes de caractères Unicode représentées par la séquence d'échappement `\p{nom}`. Leur négation `\P{nom}` existe aussi. Par exemple, pour reconnaître les caractères majuscules et minuscules :

```
use charnames ":full"; # utilisation des noms Unicode longs
$x = "BOB";
$x =~ /\p{IsUpper}/; # correspond, les caractères majuscules
$x =~ /\P{IsUpper}/; # pas de correspondance, les caractères sans majuscules
$x =~ /\p{IsLower}/; # pas de correspondance, les caractères minuscules
$x =~ /\P{IsLower}/; # correspond, les caractères sans les minuscules
```

Voici les associations entre quelques noms de classes Perl et les classes traditionnelles Unicode :

Nom de classe Perl	Nom de classe Unicode ou expression rationnelle
<code>IsAlpha</code>	<code>/^[LM]/</code>
<code>IsAlnum</code>	<code>/^[LMN]/</code>
<code>IsASCII</code>	<code>\$code <= 127</code>
<code>IsCntrl</code>	<code>/^C/</code>
<code>IsBlank</code>	<code>\$code =~ /^(0020 0009)\$/ /^Z[^\p]/</code>
<code>IsDigit</code>	<code>Nd</code>
<code>IsGraph</code>	<code>/^([LMNPS] Co)/</code>
<code>IsLower</code>	<code>Ll</code>
<code>IsPrint</code>	<code>/^([LMNPS] Co Zs)/</code>
<code>IsPunct</code>	<code>/^P/</code>
<code>IsSpace</code>	<code>/^Z/ (\$code =~ /^(0009 000A 000B 000C 000D)/</code>
<code>IsSpacePerl</code>	<code>/^Z/ (\$code =~ /^(0009 000A 000C 000D 0085 2028 2029)/</code>
<code>IsUpper</code>	<code>/^[ut]/</code>
<code>IsWord</code>	<code>/^[LMN]/ \$code eq "005F"</code>
<code>IsXDigit</code>	<code>\$code =~ /^00(3[0-9] [46][1-6])/</code>

Vous pouvez aussi utiliser les noms officiels des classes Unicode grâce à `\p` et `\P` comme dans `\p{L}` pour les 'lettres' Unicode, `\p{Lu}` pour les lettres minuscules ou `\P{Nd}` pour les non-chiffres. Si `nom` est composé d'une seule lettre, les accolades peuvent être omises. Par exemple `\pM` est la classe de caractères Unicode 'marks' pour les accents. Pour la liste complète, voir *perlunicode*.

Unicode est découpé en plusieurs sous-ensembles de caractères que vous pouvez tester par `\p{...}` (dans) et `\P{...}` (hors de). Pour tester si un caractère appartient (ou non) à un système d'écriture, vous pouvez utiliser le nom anglais de ce système, par exemple `\p{Latin}`, `\p{Greek}` ou `\P{Katakana}`. Les autres ensembles sont les blocs Unicode dont le nom commence par "In". L'un de ces blocs est dédié aux opérateurs mathématiques et on peut l'utiliser via `\p{InMathematicalOperators}`. Pour une liste complète, voir *perlunicode*.

`\X` est une abréviation pour la séquence de classes de caractères qui contient les *séquences de caractères combinatoires* Unicode. Une séquence de caractères combinatoires est un caractère de base suivi d'un certain nombre de caractères combinatoires, c'est-à-dire des signes tels que les accents qui changent la prononciation d'une lettre. En utilisant les noms Unicode longs, `A + COMBINING RING` est une séquence de caractères combinatoires avec `A` comme caractère de base et `COMBINING RING` comme caractère de combinatoire et cette séquence représente, en Danois, un `A` avec un cercle au-dessus comme dans le mot `Angstroem`. `\X` est équivalent à `\PM\pM*` qui signifie un caractère non-marqué éventuellement suivi d'une série de caractères marqués.

Pour obtenir les informations les plus récentes et complètes concernant Unicode, consultez la norme Unicode ou le site web du consortium Unicode <http://www.unicode.org/>.

Et comme si toutes ces classes ne suffisaient pas, Perl définit aussi des classes de caractères de style POSIX. Elles sont de la forme `[:nom:]` où `nom` est le nom d'une classe POSIX. Les classes POSIX sont `alpha`, `alnum`, `ascii`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper` et `xdigit` plus les deux extensions `word` (une extension Perl pour reconnaître `\w`) et `blank` (une extension GNU). Si `utf8` est actif alors ces classes sont définies de la même manière que les classes Perl Unicode correspondantes : `[:upper:]` est la même chose que `\p{IsUpper}`, etc. Les classes de caractères POSIX, par contre, ne nécessitent pas l'utilisation de `utf8`. Les classes `[:digit:]`, `[:word:]` et `[:space:]` correspondent aux classes familières `\d`, `\w` et `\s`. Pour obtenir la négation d'une classe POSIX, placez un `^` devant son nom comme dans `[:^digit:]` qui correspond à `\D` ou, avec `utf8`, à `\P{IsDigit}`. Les classes Unicode et POSIX peuvent être utilisées comme `\d`, en sachant que les classes POSIX ne sont accessibles qu'à l'intérieur d'une classe de caractères :

```

/\s+[abc[:digit:]xyz]\s*/; # reconnaît a,b,c,x,y,z ou un chiffre
/^=item\s[[:digit:]]/;    # reconnaît '=item',
                           # suivi d'un espace et d'un chiffre

use charnames ":full";
/\s+[abc\p{IsDigit}xyz]\s*/; # reconnaît a,b,c,x,y,z ou un chiffre
/^=item\s\p{IsDigit}/;      # reconnaît '=item',
                           # suivi d'un espace et d'un chiffre

```

C'est tout pour les caractères et les classes de caractères.

8.3.2 Compilation et stockage d'expressions rationnelles

Dans la partie 1, nous avons parlé du modificateur `//o` qui compile une expression rationnelle une seule fois. Cela suggère qu'une expression rationnelle compilée est une sorte de structure de données qui peut être stockée une fois et utilisée plusieurs fois. L'opérateur d'expression rationnelle `qr//` fait exactement cela : `qr/chaîne/` compile la chaîne `chaîne` en tant qu'expression rationnelle et transforme le résultat en quelque chose qui peut être affectée à une variable.

```
$reg = qr/foo+bar?/; # reg contient une expression rationnelle compilée
```

Puis `$reg` peut être utilisée en tant qu'expression rationnelle :

```

$x = "fooooba";
$x =~ $reg; # est reconnu, exactement comme /foo+bar?/
$x =~ /$reg/; # idem, sous une forme différente

```

`$reg` peut aussi être utilisée au sein d'une expression rationnelle plus grande :

```
$x =~ /(abc)?$reg/; # est encore reconnue
```

Comme avec l'opérateur de recherche de correspondances, l'opérateur d'expression rationnelle peut utiliser différents délimiteurs tels `qr!|`, `qr{}` ou `qr~`. Le délimiteur apostrophe (`qr"`) supprime l'interpolation des variables.

La pré-compilation des expressions rationnelles est utile pour des mises en correspondances dynamiques qui ne nécessitent pas une recompilation à chaque passage. En utilisant des expressions rationnelles pré-compilées, nous pouvons écrire un programme `grep_pas_a_pas` qui cherche une suite d'expressions rationnelles en passant à la suivante dès que la précédente est reconnue :

```
% cat > grep_pas_a_pas
#!/usr/bin/perl
# grep_pas_a_pas - reconnaît <num> regexps, l'une après l'autre
# usage: grep_pas_a_pas <num> regexp1 regexp2 ... file1 file2 ...

$num = shift;
$regexp[$_] = shift foreach (0..$num-1);
@compiled = map qr/$_/, @regexp;
while ($line = <>) {
    if ($line =~ /$compiled[0]/) {
        print $line;
        shift @compiled;
        last unless @compiled;
    }
}
^D

% grep_pas_a_pas 3 shift print last grep_pas_a_pas
$num = shift;
    print $line;
    last unless @compiled;
```

Le stockage des expressions rationnelles pré-compilées dans le tableau `@compiled` nous permet de faire une boucle sur les expressions rationnelles sans les recompiler. On y gagne en flexibilité sans sacrifier la vitesse.

8.3.3 Composition d'expressions rationnelles durant l'exécution

Les retours arrières sont plus efficaces que des essais successifs avec plusieurs expressions rationnelles. Si vous avez plusieurs expressions rationnelles et que la reconnaissance d'une seule d'entre elles est acceptable alors il est possible de les combiner sous la forme d'une alternative. Si les expressions initiales sont des données d'entrée, cela peut être fait via une opération de jointure. Nous allons exploiter cette idée dans une version améliorée du programme `simple_grep`, un programme qui reconnaît plusieurs motifs :

```
% cat > multi_grep
#!/usr/bin/perl
# multi_grep - reconnaît une regexp parmi <num> regexps
# usage: multi_grep <num> regexp1 regexp2 ... file1 file2 ...

$num = shift;
$regexp[$_] = shift foreach (0..$num-1);
$pattern = join '|', @regexp;

while ($line = <>) {
    print $line if $line =~ /$pattern/o;
}
^D

% multi_grep 2 shift for multi_grep
$num = shift;
$regexp[$_] = shift foreach (0..$num-1);
```

Il est parfois avantageux de construire le motif à partir de l'entrée qui doit être analysée et d'utiliser les valeurs autorisées dans la partie gauche de l'opérateur de reconnaissance. Comme exemple de cette situation apparemment paradoxale, nous allons supposer que notre entrée contient une commande qui doit correspondre à l'une des commandes autorisées mais en ajoutant que cette commande peut être abrégée s'il n'y a pas d'ambiguïté possible avec une autre commande. Le programme ci-dessous illustre l'algorithme de base :

```
% cat > keymatch
#!/usr/bin/perl
$kwds = 'copy compare list print';
while( $command = <> ){
    $command =~ s/^\s+|\s+$//g; # trim leading and trailing spaces
    if( ( @matches = $kwds =~ /\b$command\w*/g ) == 1 ){
        print "command: '$matches'\n";
    } elsif( @matches == 0 ){
        print "no such command: '$command'\n";
    } else {
        print "not unique: '$command' (could be one of: @matches)\n";
    }
}
^D

% keymatch
li
command: 'list'
co
not unique: 'co' (could be one of: copy compare)
printer
no such command: 'printer'
```

Plutôt que d'essayer de reconnaître les commandes existantes à l'intérieur de l'entrée fournie, nous essayons de reconnaître cette entrée dans l'ensemble des commandes. L'opération de mise en correspondance `$kwds =~ /\b($command\w*)/g` fait plusieurs choses à la fois. Elle s'assure que la commande fournie commence bien là où un mot clé débute (`\b`). Elle accepte les abréviations grâce à `\w*`. Elle indique le nombre de correspondances (scalar `@matches`) et donne tous les mots clés réellement reconnus. Vous pouvez difficilement demander plus.

8.3.4 Commentaires et modificateurs intégrés dans une expression rationnelle

À partir d'ici, nous allons parler des *motifs étendus* de Perl. Ce sont des extensions de la syntaxe traditionnelle des expressions rationnelles qui fournissent de nouveaux outils utiles pour la recherche de motifs. Nous avons déjà vu des extensions telles que les quantificateurs minimaux `??`, `*?`, `+?`, `{n,m}?` et `{n,}?`. Les autres extensions sont toutes de la forme `(?car...)` où `car` est un caractère qui détermine le type de l'extension.

La première extension est un commentaire `(?#texte)`. Cela permet d'inclure un commentaire dans l'expression rationnelle sans modifier sa signification. Le commentaire ne doit pas contenir de parenthèse fermante dans son texte. Un exemple :

```
/(?# reconnaît un entier:)[+-]?\d+;/
```

Ce style de commentaires a été largement amélioré grâce au modificateur `//x` qui permet des commentaires beaucoup plus libres et simples.

Les modificateurs `//i`, `//m`, `//s` et `//x` (ou une combinaison de plusieurs d'entre eux) peuvent eux aussi être intégrés dans une expression rationnelle en utilisant `(?i)`, `(?m)`, `(?s)` et `(?x)`. Par exemple :

```
/(?i)yes/; # reconnaît 'yes' sans tenir compte de la casse
/yes/i;    # la même chose
/(?x)(
    [+-]? # un signe optionnel
    \d+   # les chiffres
)
/x;
```

Les modificateurs inclus ont deux avantages importants sur les modificateurs habituels. Tout d'abord, ils permettent de spécifier une combinaison de modificateurs différente pour *chaque* partie de l'expression rationnelle. C'est pratique pour mettre en correspondance un tableau d'expressions rationnelles qui ont des modificateurs différents :

```
$pattern[0] = '(?i)doctor';
$pattern[1] = 'Johnson';
...
while (<>) {
    foreach $patt (@pattern) {
        print if /$patt/;
    }
}
```

Ensuite parce que les modificateurs inclus n'agissent que sur le groupe dans lequel ils apparaissent (sauf //p qui modifie l'expression rationnelle dans son ensemble). Donc, le regroupement peut être utilisé pour «localiser» l'effet des modificateurs :

```
/Answer: ((?i)yes)/; # reconnaît 'Answer: yes', 'Answer: YES', etc.
```

Les modificateurs inclus peuvent aussi annuler des modificateurs déjà présents en utilisant par exemple (?-i). Les modificateurs peuvent être combinés en une seule expression comme (?s-i) qui active le mode simple ligne et annule l'insensibilité à la casse.

Les modificateurs inclus sont aussi utilisables dans un regroupement sans mémorisation. (?i-m:regexp) est un regroupement sans mémorisation qui reconnaît `regexp` sans être sensible à la casse et en désactivant le mode multi-lignes.

8.3.5 Regarder en arrière et regarder en avant

Cette section présente les assertions permettant de regarder en arrière ou en avant. Tout d'abord quelques petits éléments d'introduction.

Dans les expressions rationnelles en Perl, la plupart des éléments 'consomment' une certaine quantité de la chaîne avec lesquels ils sont mis en correspondance. Par exemple, l'élément [abc] consomme un caractère de la chaîne lorsqu'il est reconnu, dans le sens où Perl avance au caractère suivant dans la chaîne après la mise en correspondance. Il existe certains éléments, par contre, qui ne consomment aucun caractère (il ne change pas la position) lorsqu'ils sont reconnus. Les exemples que nous avons déjà vus sont des ancres. L'ancre ^ reconnaît le début de ligne mais ne consomme aucun caractère. De même, l'ancre de bordure de mot \b est reconnue entre un caractère qui est reconnu par \w et un caractère qui ne l'est pas mais elle ne consomme aucun caractère. Ces ancres sont des exemples d'assertions de longueur nulle. « De longueur nulle » parce qu'elles ne consomment aucun caractère et « assertion » parce qu'elles testent une propriété de la chaîne. Dans le contexte de notre analogie avec la traversée d'une forêt, la plupart des éléments des expressions rationnelles nous font avancer le long du chemin alors que les ancres nous arrêtent pour regarder autour de nous. Si l'environnement local convient, nous pouvons continuer. Sinon, il faut effectuer un retour arrière.

Vérifier l'environnement peut impliquer un regard en avant, en arrière ou les deux. ^ regarde en arrière pour vérifier qu'il n'y a aucun caractère avant. \$ regarde en avant pour vérifier qu'il n'y a pas de caractère après. \b regarde en avant et en arrière pour vérifier que les deux caractères qui l'entourent sont de natures différentes (mot et non-mot).

Les assertions en avant et en arrière sont des généralisations du concept d'ancres. Ce sont des assertions de longueur nulle qui nous permettent de spécifier les caractères que nous voulons. L'assertion de regard en avant est notée (?=regexp) alors que l'assertion de regard en arrière est notée (?<=fixed-regexp). Voici quelques exemples :

```
$x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(?=\s)/; # reconnaît 'cat' dans 'housecat'
@catwords = ($x =~/(?<=\s)cat\w+/g); # correspond avec
# $catwords[0] = 'catch'
# $catwords[1] = 'catnip'
$x =~ /\bcat\b/; # reconnaît 'cat' dans 'Tom-cat'
$x =~/(?<=\s)cat(?=\s)/; # pas de correspondance; pas de 'cat' isolé
# au milieu de $x
```

Notez que les parenthèses dans `(?=regex)` et `(?<=regex)` sont sans mémorisation puisque ce sont des assertions de longueur nulle. Donc, dans le deuxième exemple, les sous-chaînes capturées sont celles reconnues par l'expression rationnelle complète. Les assertions en avant `(?=regex)` peuvent reconnaître une expression rationnelle quelconque. Par contre, les assertions en arrière `(?<=fixed-regex)` ne fonctionnent que pour des expressions rationnelles de longueur fixe (dont le nombre de caractères est fixé). Donc `(?<=(ab|bc))` est correct mais pas `(?<=(ab)*)`. Les négations de ces assertions se notent respectivement `(?!regex)` et `(?<!\fixed-regex)`. Elles sont évaluées à vrai si l'expression rationnelle ne correspond *pas* :

```
$x = "foobar";
$x =~ /foo(?!bar)/; # pas de correspondance, 'bar' suit 'foo'
$x =~ /foo(?!baz)/; # reconnue, 'baz' ne suit pas 'foo'
$x =~ /(?!\s)foo/; # reconnue, il n'y a pas de \s avant 'foo'
```

La classe `\C` n'est pas utilisable dans une assertion en arrière parce que cette classe utilise une tricherie sur la définition d'un caractère qui le deviendrait encore plus en regardant vers l'arrière.

Voici un exemple où une chaîne doit être découpée en retrouvant chaque mot et chaque nombre séparés par des espaces et chacun des tirets considérés isolément. Si on tente d'utiliser `/\s+/,` cela ne marche pas car les espaces ne sont pas nécessaires entre les tirets ou entre un mot et un tiret. Les points de découpe supplémentaire sont identifiés en regardant en avant et en arrière.

```
$str = "one two - --6-8";
@toks = split / \s+          # un groupe d'espaces
        | (?<=\S) (?=-)     # un non-espace suivi d'un '-'
        | (?<=-) (?=\S)    # un '-' suivi d'un non-espace
        /x, $str;          # @toks = qw(one two - - - 6 - 8)
```

8.3.6 Utilisation de sous-expressions indépendantes pour empêcher les retours arrière

Les *sous-expressions indépendantes* sont des expressions rationnelles qui, dans le contexte d'une expression rationnelle plus grande, fonctionnent indépendamment de cette expression rationnelle globale. C'est à dire qu'elles consomment tout ce qu'elles veulent de la chaîne sans prendre en compte la possibilité de reconnaissance de l'expression rationnelle globale. Les sous-expressions indépendantes sont représentées par `(?>regex)`. Pour illustrer leur comportement, considérons tout d'abord une expression rationnelle ordinaire :

```
$x = "ab";
$x =~ /a*ab/; # correspondance
```

Il y a évidemment correspondance mais lors de la reconnaissance, la sous-expression `a*` consomme d'abord le `a`. Ce faisant, elle empêche la reconnaissance de l'expression rationnelle globale. Donc, après retour arrière, `a*` laisse le `a` et reconnaît la chaîne vide. Ici, ce que `a*` a reconnu est *dépendant* de ce qui est reconnu par le reste de l'expression rationnelle.

En revanche, considérons l'expression rationnelle avec une sous-expression indépendante :

```
$x =~ /(?!>a*)ab/; # pas de correspondance
```

La sous-expression indépendante `(?!>a*)` ne tiens pas compte du reste de l'expression rationnelle et donc consomme le `a`. Le reste de l'expression rationnelle `ab` ne peut plus trouver de correspondances. Puisque `(?!>a*)` est indépendante, il n'y a pas de retour arrière et la sous-expression indépendante ne relâche pas son `a`. Donc l'expression rationnelle globale n'est pas reconnue. On observe un comportement similaire avec deux expressions rationnelles complètement indépendantes :

```
$x = "ab";
$x =~ /a*/g; # est reconnue, consomme le 'a'
$x =~ /\Gab/g; # pas de correspondance, plus de 'a' disponible
```

Ici `//g` et `\G` créent un point de non-retour entre les deux expressions rationnelles. Les expressions rationnelles avec des sous-expressions indépendantes font un peu la même chose en plaçant un point de non-retour après la reconnaissance d'une sous-expression indépendante.

Cette possibilité de blocage du retour arrière grâce aux sous-expressions indépendantes est très pratique. Supposons que nous voulons reconnaître une chaîne non-vide entre parenthèses (pouvant contenir elle-même un niveau de parenthèses). L'expression rationnelle suivante fonctionnera :

```
$x = "abc(de(fg)h)"; # parenthèse non refermée
$x =~ /\( ( [^()]+ | \( [^()]* \) )+ \)/x;
```

L'expression rationnelle reconnaît une parenthèse ouvrante, une ou plusieurs occurrences d'une alternative et une parenthèse fermante. La première branche de l'alternative `[^()]+` reconnaît un sous-chaîne sans parenthèse et la seconde branche `\([^\(\)]*\)` reconnaît une sous-chaîne entourée de parenthèses. Cette expression rationnelle est malheureusement pathologique : elle contient des quantificateurs imbriqués de la forme `(a+|b)+`. Nous avons expliqué dans la partie 1 pourquoi une telle imbrication impliquait un temps d'exécution exponentiel en cas de non-reconnaissance. Pour prévenir cette explosion combinatoire, nous devons empêcher les retours arrière inutiles quelque part. On peut y arriver en incluant le quantificateur interne dans une sous-expression indépendante :

```
$x =~ /\( ( (?>[^()]+) | \( [^()]* \) )+ \)/x;
```

Ici, `(?>[^()]+)` interrompt le partitionnement combinatoire de la chaîne en consommant la partie la plus grande possible de la chaîne sans permettre le retour arrière. Donc, la non-reconnaissance sera détectée beaucoup plus rapidement.

8.3.7 Les expressions conditionnelles

Une *expression conditionnelle* est une forme d'instruction si-alors-sinon qui permet de choisir entre deux motifs à reconnaître selon une condition. Il existe deux types d'expressions conditionnelles : `(?(condition)motif-oui)` et `(?(condition)motif-oui|motif-non)`. `(?(condition)motif-oui)` agit comme une instruction 'if () {}' en Perl. Si la condition est vraie, le motif-oui doit être reconnu. Si la condition est fausse, le motif-oui n'est pas pris en compte et Perl passe à l'élément suivant dans l'expression rationnelle. La seconde forme est comme une instruction 'if () {} else {}' en Perl. Si la condition est vraie le motif-oui doit être reconnu sinon c'est le motif-non qui doit être reconnu.

La condition a plusieurs formes possibles. La première forme est simplement un entier entre parenthèses (*entier*). La condition est vraie si le regroupement mémorisé correspondant `\entier` a été reconnue plus tôt dans l'expression rationnelle. On peut faire la même chose avec un nom associé à un regroupement mémorisé en écrivant soit `(<nom>)` soit `('nom')`. La seconde forme est une assertion de longueur nulle `(?. . .)`, soit en avant, soit en arrière, soit une assertion de code (dont on parlera dans la section suivante). La troisième forme fournit des tests qui retournent vrai si l'expression est évaluée à l'intérieur d'une récursion `((R))` ou est appelée depuis une regroupement mémorisé référencé par son numéro `((R1), (R2),...)` ou par son nom `((R&nom))`.

La forme de la condition utilisant un entier ou un nom nous permet de choisir, avec plus de flexibilité, ce que l'on veut reconnaître en fonction de ce qui a déjà été reconnu. L'exemple suivant cherche les mots de la forme `"xy$z"` ou `"$xyy$z"` :

```
% simple_grep '^(\\w+)(\\w+)?(?(2)\\2\\1|\\1)$' /usr/dict/words
beriberi
coco
couscous
deed
...
toot
toto
tutu
```

La condition sous la forme d'une assertion de longueur nulle en arrière, combinée avec des références arrières, permet à une partie de la reconnaissance d'influencer une autre partie de la reconnaissance qui arrive plus tard. Par exemple :

```
/[ATGC]+(?(?<=AA)G|C)/;
```

reconnaît une séquence d'ADN qui ne se termine ni par AAG ni par C. Remarquez la notation `(?(?<=AA)G|C)` et non pas `(?(?<=AA))G|C` ; pour les assertions de longueur nulle, les parenthèses ne sont pas nécessaires.

8.3.8 Définir des motifs nommés

Certaines expressions rationnelles doivent réutiliser plusieurs fois le même sous-motif en différents endroits. Depuis Perl 5.10, il est possible de définir des sous-motifs nommés afin d'y faire appel n'importe où dans l'expression rationnelle les contenant. La syntaxe pour définir un tel groupe est `(?(DEFINE)(?<nom>motif)...) . L'appel d'un tel groupe s'écrit (?&nom).`

L'exemple ci-dessous illustre cette fonctionnalité en s'appuyant sur le motif des nombres à virgule flottante présenté précédemment. Les trois sous-motifs utilisés plus d'une fois sont le signe optionnel (`osg`), la suite de chiffres pour un entier (`int`) et la partie décimale (`dec`). Le groupe `DEFINE` à la fin contient leur définition. Notez que le motif pour la partie décimale est le premier endroit où nous pouvons réutiliser le motif pour un entier.

```
/^ (?&osg)\ * ( (?&int)(?&dec)? | (?&dec) )
(?: [eE](?&osg)(?&int) )?
$
?(DEFINE)
  (?<osg>[-+]?          # signe optionnel
  (?<int>\d++          # entier
  (?<dec>\.(?&int))    # partie décimale
)/x
```

8.3.9 Motifs récursifs

Cette fonctionnalité (apparue dans Perl 5.10) augmente beaucoup la puissance des expressions rationnelles de Perl. En se référant à un regroupement avec mémorisation via la construction `(?groupe-ref)`, le *motif* du regroupement référencé est utilisé comme un sous-motif indépendant à l'endroit où est placée la référence. Comme cette référence peut apparaître *dans* le regroupement auquel elle fait référence, cela rend possible l'utilisation d'expressions rationnelles pour des tâche qui, jusqu'ici, nécessitait un analyseur récursif.

Pour illustrer cette fonctionnalité, nous allons construire un motif qui reconnaît un palindrome (c'est un mot ou une phrase qui, en faisant abstraction des espaces et de la ponctuation, se lit de la même manière à l'endroit et à l'envers). Nous commençons par constater qu'une chaîne vide ou ne contenant qu'un seul caractère est un palindrome. Sinon elle doit avoir un caractère mot au début, ce même caractère mot à la fin et un palindrome entre les deux.

```
/(?: (\w) (?...ici un palindrome...) \{-1\} | \w? )/x
```

En ajoutant `\W*` aux deux extrémités pour supprimer ce qui peut être ignoré, on obtient le motif complet :

```
my $pp = qr/^(\\W* (?: (\\w) (?1) \\g{-1} | \\w? ) \\W*)$/ix;
for $s ( "saippuakauppias", "A man, a plan, a canal: Panama!" ){
  print "'$s' is a palindrome\n" if $s =~ /$pp/;
}
```

Dans la construction `(?...)`, on peut utiliser une référence absolue ou relative. Le motif complet peut être réinséré via `(?R)` ou `(?0)`. Si vous préférez nommer vos regroupements, vous pouvez aussi utiliser `(?&nom)` pour les utiliser récursivement.

8.3.10 Un peu de magie : exécution de code Perl dans une expression rationnelle

Normalement les expressions rationnelles sont une partie d'une expression Perl. Les expressions d'évaluation de code renversent cela en permettant de placer n'importe quel code Perl à l'intérieur d'une expression rationnelle. Une expression d'évaluation de code est notée `(?{code})` où *code* est une chaîne contenant des instructions Perl.

Notez que cette fonctionnalité est encore considérée comme expérimentale et peut donc changer sans avertissement.

Une expression d'évaluation de code est une assertion de longueur nulle et la valeur qu'elle retourne dépend de son environnement. Il y a deux possibilités : soit l'expression est utilisée comme condition dans une expression conditionnelle `(?(condition)...) .` soit ce n'est pas le cas. Si l'expression d'évaluation de code est une condition, le code est évalué et son résultat (c'est à dire le résultat de la dernière instruction) est utilisé pour déterminer la véracité de la condition. Si l'expression d'évaluation de code n'est pas une condition, l'assertion est toujours vraie et le résultat du code est stocké dans la variable spéciale `$^R`. Cette variable peut être utilisée dans des expressions d'évaluation de code apparaissant plus tard dans l'expression rationnelle. Voici quelques exemples :


```

$x = "abcdef";
$x =~ /abc(?:print "Hi Mom!";)def/; # reconnue,
                                     # affiche 'Hi Mom!'
$x =~ /aaa(?:print "Hi Mom!";)def/; # non reconnue,
                                     # pas de 'Hi Mom!'

```

Attention à l'exemple suivant :

```

$x =~ /abc(?:print "Hi Mom!";)ddd/; # non reconnue,
                                     # pas de 'Hi Mom!'
                                     # mais pourquoi ?

```

En première approximation, vous pourriez croire qu'il n'y a pas d'affichage parce que ddd ne peut être reconnu dans la chaîne explorée. Mais regardez l'exemple qui suit :

```

$x =~ /abc(?:print "Hi Mom!";)[d]dd/; # non reconnue,
                                     # mais _affiche_ 'Hi Mom!'

```

Que s'est-il passé dans ce cas ? Si vous avez tout suivi jusqu'ici, vous savez que les deux expressions rationnelles précédentes sont équivalentes – enfermer le d dans une classe de caractères ne change en rien ce qui peut être reconnu. Alors pourquoi l'une n'affiche rien alors que l'autre le fait ?

La réponse est liée aux optimisations faites par le moteur d'expressions rationnelles. Dans le premier cas, tout ce que le moteur voit c'est une série de caractères simples (mis à part la construction `print`). Il est assez habile pour s'apercevoir que la chaîne 'ddd' n'apparaît pas dans la chaîne explorée avant même de commencer son exploration. Alors que dans le second cas, nous avons utilisé une astuce pour lui faire croire que notre motif était plus compliqué qu'il ne l'est. Il voit donc notre classe de caractères et décide qu'il doit commencer l'exploration pour déterminer si notre expression rationnelle peut ou non être reconnue. Et lors de cette exploration, il atteint l'instruction d'affichage avant de s'apercevoir qu'il n'y a pas de correspondance possible.

Pour en savoir un peu plus sur les optimisations faites par le moteur, reportez-vous à la section Directives (pragma) et déverminage (§8.3.12) plus bas.

D'autres exemples avec `print` :

```

$x =~ /(?:print "Hi Mom!";)/;          # reconnue,
                                     # affiche 'Hi Mom!'
$x =~ /(?:$c = 1;)(?:print "$c";)/;   # reconnue,
                                     # affiche '1'
$x =~ /(?:$c = 1;)(?:print "$^R";)/;  # reconnue,
                                     # affiche '1'

```

La magie évoquée dans le titre de cette section apparaît lorsque le processus de recherche de correspondance effectue un retour arrière. Si le retour arrière implique une expression d'évaluation de code et si les variables modifiées par ce code ont été localisées (via `local`) alors les modifications faites sur ces variables sont annulées ! Donc, si vous voulez compter le nombre de fois ou un caractère à été reconnu lors d'une mise en correspondance, vous pouvez utiliser le code suivant :

```

$x = "aaaa";
$count = 0; # initialisation du compteur de 'a'
$c = "bob"; # pour montrer que $c n'est pas modifié
$x =~ /(?:local $c = 0;)/;          # initialisation du compteur
    ( a                               # 'a' est reconnu
      (?:local $c = $c + 1;)/;       # incrémentation du compteur
    )*                               # on répète cela autant que possible
    aa                               # mais on reconnaît 'aa' à la fin
    (?:$count = $c;)/;              # recopie de $c local dans $count
/x;
print "'a' count is $count, \ $c variable is '$c'\n";

```

Ce code affiche :

```
'a' count is 2, $c variable is 'bob'
```

Si nous remplaçons `(?{local $c = $c + 1;})` par `(?{$c = $c + 1;})`, les modifications faites à la variable ne sont pas annulées lors du retour arrière et nous obtenons :

```
'a' count is 4, $c variable is 'bob'
```

Notez bien que seules les modifications aux variables localisées sont annulées. Les autres effets secondaires de l'exécution du code sont permanents. Donc :

```
$x = "aaaa";
$x =~ /(a(?{print "Yow\n";}))*aa/;
```

produit :

```
Yow
Yow
Yow
Yow
```

Le résultat `^R` est automatiquement localisés et donc il se comporte bien lors de retours arrière.

Cet exemple utilise une expression d'évaluation de code dans une condition pour reconnaître un article défini, 'the' pour l'anglais ou 'der|die|das' pour l'allemand :

```
$lang = 'DE'; # en allemand
...
$text = "das";
print "matched\n"
  if $text =~ /(?(?{
    $lang eq 'EN'; # est-on en anglais ?
  })
  the |          # si oui, alors il faut reconnaître 'the'
  (der|die|das) # sinon, reconnaître 'die|das|der'
)
/xi;
```

Remarquez ici que la syntaxe est `(?(?{...})motif-oui|motif-non)` et non pas `(?(?{...})motif-oui|motif-non)`. En d'autres termes, dans le cas d'une expression d'évaluation de code, les parenthèses autour de la condition sont optionnelles.

Si vous tentez d'utiliser des expressions d'évaluation de code combinées avec des variables interpolées, Perl risque de vous surprendre :

```
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ $bar })bar/; # se compile bien, $bar n'est pas interpolée
/foo(?{ 1 })$bar/;  # erreur de compilation !
/foo${pat}bar/;     # erreur de compilation !

$pat = qr/(?{ $foo = 1 })/; # précompilation d'une expression
/foo${pat}bar/;          # se compile bien
```

Si une expression rationnelle contient soit des expressions d'évaluation de code et des variables interpolées, soit une variable dont l'interpolation donne une expression d'évaluation de code, Perl traite cela comme une erreur dans l'expression rationnelle. En revanche, si l'expression d'évaluation de code est précompilée dans une variable, l'interpolation fonctionne. Pourquoi cette erreur ?

C'est parce que la combinaison de l'interpolation de variables et des expressions d'évaluation de code est risquée. Elle est risquée parce que beaucoup de programmeurs qui écrivent des moteurs de recherche utilisent directement les valeurs fournies par l'utilisateur dans leurs expressions rationnelles :

```
$regexp = <>; # lecture de l'expression rationnelle de l'utilisateur
$chomp $regexp; # suppression d'un éventuel passage à la ligne
$text =~ /$regexp/; # recherche de $regexp dans $text
```

Si la variable `$regexp` pouvait contenir des expressions d'évaluation de code, l'utilisateur pourrait exécuter n'importe quel code Perl. Par exemple, un petit rigolo pourrait chercher `system('rm -rf *')`; pour effacer tous vos fichiers. En ce sens, la combinaison de l'interpolation de variables et des expressions d'évaluation de code *souillerait* votre expression rationnelle. Donc, par défaut, cette combinaison n'est pas autorisée. Si vous n'êtes pas concerné par les utilisateurs malicieux, il est possible de désactiver cette interdiction en invoquant `use re 'eval'` :

```
use re 'eval';      # pas de sécurité
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ 1 })$bar/; # se compile bien
/foo${pat}bar/;   # se compile bien
```

Une autre forme d'expressions impliquant une évaluation de code est l'*expression d'évaluation de code produisant un motif*. Cette expression est comme la précédente sauf que le résultat de l'évaluation du code est traité comme un élément d'expression rationnelle à mettre en correspondance immédiatement. Voici un exemple simple :

```
$length = 5;
$char = 'a';
$x = 'aaaaabb';
$x =~ /(??{$char x $length})/x; # reconnue, il y a bien 5 'a'
```

Le dernier exemple contient à la fois des expressions ordinaires et des expressions impliquant de l'évaluation de code. Il détecte si les espacements entre les 1 d'une chaîne binaire `1101010010001...` respectent une suite de Fibonacci 0, 1, 1, 2, 3, 5...

```
$x = "1101010010001000001";
$s0 = 0; $s1 = 1; # conditions initiales
print "C'est une suite de Fibonacci\n"
  if $x =~ /^1          # on reconnaît le '1' initial
      (?:
          ((??{ $z0 })) # match some '0'
          1             # and then a '1'
          (?{ $z0 = $z1; $z1 .= $^N; })
      )+ # répété autant que nécessaire
      $      # c'est la fin
  /x;
printf "La séquence la plus large mesure %d\n",
  length($z1)-length($z0);
```

Souvenez-vous que `$^N` contient ce qui a été reconnu par le dernier regroupement complètement reconnu. Cela affiche :

```
C'est une suite de Fibonacci
La séquence la plus large mesure 5
```

Remarquez que les variables `$z0` et `$z1` ne sont pas interpolées lorsque l'expression rationnelle est compilée comme cela se passerait pour des variables en dehors d'une expression d'évaluation de code. En fait, le code est évalué à chaque fois que Perl le rencontre lors de la mise en correspondance.

Cette expression rationnelle sans le modificateur `//x` est :

```
/^1(?:((??{ $z0 }))1(?{ $z0 = $z1; $z1 .= $^N; }))+$/
```

Cela montre que l'usage d'espaces dans la partie code est encore possible. Néanmoins lorsqu'on travaille avec du code et des expressions conditionnelles, la forme étendue des expressions rationnelles est quasiment toujours nécessaire pour en faciliter la création et la maintenance.

8.3.11 Ordres de contrôle du retour-arrière

Perl 5.10 propose un certain nombre d'ordres de contrôle permettant de piloter finement le processus de retour-arrière en influençant directement le moteur d'expressions rationnelles et en fournissant des moyens de surveillance. Comme d'habitude ces nouvelles fonctionnalités sont expérimentales et sujettent à changement voire suppression dans de futures versions de Perl. Le lecteur intéressé pourra se référer à *Special Backtracking Control Verbs* in *perlre* pour une description plus détaillée.

Ci-dessous vous trouverez un simple exemple qui illustre l'usage de l'ordre (`*FAIL`) qui s'abrège en (`*F`). Si cet ordre est inséré dans une expression rationnelle, il la fera échouer exactement comme le ferait une non correspondance entre la chaîne et le motif recherché. Le traitement de l'expression rationnelle continue comme après un échec "normal" et donc, par exemple, le moteur essaye une autre branche d'une alternative ou avance à une position suivante dans la chaîne. Comme pour un échec ne préserve pas les groupes ou les résultats produits, il peut être nécessaire de coupler cet ordre avec du code embarqué.

```
%count = ();
"supercalifragilisticexpialidoceous" =~
  /([aeiou])(?{ $count{$1}++; })(*FAIL)/oi;
printf "%3d '%s'\n", $count{$_}, $_ for (sort keys %count);
```

Le motif commence par une classe de caractères contenant un sous-ensemble des lettres. À chaque fois que cette classe est reconnue, l'instruction `$count{'a'}++`; est exécutée pour incrémenter les compteurs de lettres. Puis l'ordre (`*FAIL`) est rencontré et le moteur d'expression rationnelle effectue ce qu'on lui demande : tant qu'il ne rencontre pas la fin de la chaîne, il avance dans la chaîne à la recherche d'une autre voyelle. Donc, qu'il y ait ou non reconnaissance importe peu : le moteur travaille jusqu'à l'inspection complète de la chaîne. Il faut noter que la solution suivante :

```
$count{lc($_)}++ for split(' ', "supercalifragilisticexpialidoceous");
printf "%3d '%s'\n", $count2{$_}, $_ for (qw{ a e i o u } );
```

est considérablement plus lente.

8.3.12 Directives (pragma) et déverminage

Il existe plusieurs directives (pragma) pour contrôler et déboguer les expressions rationnelles en Perl. Nous avons déjà rencontré une directive dans la section précédente, `use re 'eval'`; qui autorise la coexistence de variables interpolées et d'expressions d'évaluation de code dans la même expression rationnelle. Les autres directives sont :

```
use re 'taint';
$tainted = <>;
@parts = ($tainted =~ /(\w+)\s+(\w+)/; # @parts est maintenant souillé
```

La directive `taint` rend souillées les sous-chaînes extraites lors d'une mise en correspondance appliquée à une chaîne souillée. Ce n'est pas le cas par défaut puisque les expressions rationnelles sont souvent utilisées pour extraire une information sûre d'une chaîne souillée. Utilisez `taint` lorsque l'extraction ne garantit pas la sûreté de l'information extraite. Les deux directives `taint` et `eval` ont une portée lexicale limitée au bloc qui les englobe.

```
use re 'debug';
/^(.*)$/s;      # affichage d'information de debug

use re 'debugcolor';
/^(.*)$/s;      # affichage d'information de debug en couleur
```

Les directives globales `debug` et `debugcolor` vous permettent d'obtenir des informations détaillées lors de la compilation et de l'exécution des expressions rationnelles. `debugcolor` est exactement comme `debug` sauf que les informations sont affichées en couleur sur les terminaux qui sont capables d'afficher les séquences `termcap` de colorisation. Voici quelques exemples de sorties :

```
% perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
Compiling REX 'a*b+c'
size 9 first at 1
  1: STAR(4)
  2:  EXACT <a>(0)
  4: PLUS(7)
  5:  EXACT <b>(0)
  7: EXACT <c>(9)
  9: END(0)
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REX 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
Matching REX 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
  0 <> <abc>          | 1: STAR
                        EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
  1 <a> <bc>          | 4:  PLUS
                        EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
  2 <ab> <c>          | 7:    EXACT <c>
  3 <abc> <>          | 9:    END
Match successful!
Freeing REX: 'a*b+c'
```

Si vous êtes arrivés aussi loin dans ce tutoriel, vous pouvez probablement comprendre à quoi correspondent les différentes parties de cette sortie. La première partie :

```
Compiling REX 'a*b+c'
size 9 first at 1
  1: STAR(4)
  2:  EXACT <a>(0)
  4: PLUS(7)
  5:  EXACT <b>(0)
  7: EXACT <c>(9)
  9: END(0)
```

décrit la phase de compilation. STAR(4) signifie qu'il y a un objet étoilé, dans ce cas 'a', et qu'une fois reconnu, il faut aller en 4, c'est à dire PLUS(7). Les lignes du milieu décrivent des heuristiques et des optimisations appliquées avant la mise en correspondance :

```
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REX 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
```

Puis la mise en correspondance est effectuée et les lignes qui restent décrivent ce processus :

```
Matching REX 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
  0 <> <abc>          | 1: STAR
                        EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
  1 <a> <bc>          | 4:  PLUS
                        EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
  2 <ab> <c>          | 7:    EXACT <c>
  3 <abc> <>          | 9:    END
Match successful!
Freeing REX: 'a*b+c'
```

Chaque pas est de la forme `n <x> <y>` où `<x>` est la partie de la chaîne reconnue et `<y>` est la partie de la chaîne non encore reconnue. | 1: STAR indique que Perl est rendu à la ligne 1 de la phase de compilation vue précédemment. Voir *Débogage des expressions rationnelles* in *perldebguts* pour de plus amples informations.

Une autre méthode pour déboguer les expressions rationnelles consiste à inclure des instructions `print` dans l'expression rationnelle.

```
"that this" =~ m@(?{print "Start at position ", pos, "\n";})
    t(?{print "t1\n";})
    h(?{print "h1\n";})
    i(?{print "i1\n";})
    s(?{print "s1\n";})
    |
    t(?{print "t2\n";})
    h(?{print "h2\n";})
    a(?{print "a2\n";})
    t(?{print "t2\n";})
    (?{print "Done at position ", pos, "\n";})
@x;
```

qui affiche :

```
Start at position 0
t1
h1
t2
h2
a2
t2
Done at position 4
```

8.4 BUGS

Les expressions d'évaluation de code, les expressions conditionnelles et les expressions indépendantes sont *expérimentales*. Ne les utilisez pas dans du code de production. Pas encore.

8.5 VOIR AUSSI

Ce document n'est qu'un simple tutoriel. Pour une documentation complète sur les expressions rationnelles en Perl, voir *perltre*.

Pour plus d'informations sur l'opérateur de mise en correspondance `m//` et l'opérateur de substitution `s//`, voir *Opérateurs d'expression rationnelle* in *perlop*. Pour plus d'informations sur les opérations de découpage via `split`, voir `split` in *perlfunc*.

Un très bon livre sur l'utilisation des expressions rationnelles, voir le livre *Mastering Regular Expressions* par Jeffrey Friedl (édité par O'Reilly, ISBN 1556592-257-3) (N.d.t: une traduction française existe.)

8.6 AUTEURS ET COPYRIGHT

Copyright (c) 2000 Mark Kvale Tous droits réservés.

Ce document peut être distribué sous les mêmes conditions que Perl lui-même.

8.6.1 Remerciements

L'exemple des codons dans une chaîne d'ADN est librement inspiré de l'exemple de codes ZIP du chapitre 7 de *Mastering Regular Expressions*.

L'auteur tient à remercier Jeff Pinyan, Andrew Johnson, Peter Haworth, Ronald J Kimball et Joe Smith pour leur aide et leurs commentaires.

8.7 TRADUCTION

8.7.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

8.7.2 Traducteur

Traduction initiale et mise à jour vers 5.10.0 : Paul Gaborit `Paul.Gaborit at enstimac.fr`.

8.7.3 Relecture

Jean-Louis Morel <jl_morel@bribes.org>

Chapitre 9

perlboot

Tutoriel pour l'orienté objet à destination des débutants

9.1 DESCRIPTION

Si vous n'êtes pas familier avec les notions de programmation orientée objet d'autres langages, certaines documentations concernant les objets en Perl doivent vous sembler décourageantes. Ces documents sont *perlobj*, la référence sur l'utilisation des objets, et *perltoot* qui présente, sous forme de tutoriel, les particularités des objets en Perl.

Ici nous utiliserons une approche différente en supposant que vous n'avez aucune expérience préalable avec l'objet. Il est quand même souhaitable de connaître les sous-routines (*perlsub*), les références (*perlref* et autres) et les paquetages (*perlmod*). Essayez de vous familiariser avec ces concepts si vous ne l'avez pas déjà fait.

9.1.1 Si nous pouvions parler aux animaux...

Supposons un instant que les animaux parlent :

```
sub Boeuf::fait {
    print "un Boeuf fait mheuu !\n";
}
sub Cheval::fait {
    print "un Cheval fait hiiii !\n";
}
sub Mouton::fait {
    print "un Mouton fait bêêê !\n"
}

Boeuf::fait;
Cheval::fait;
Mouton::fait;
```

Le résultat sera :

```
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
```

Ici, rien de spectaculaire. De simples sous-routines, bien que dans des paquetages séparés, appelées en utilisant leur nom complet (incluant le nom du paquetage). Créons maintenant un troupeau :

```
# Boeuf::fait, Cheval::fait, Mouton::fait comme au-dessus
@troupeau = qw(Boeuf Boeuf Cheval Mouton Mouton);
foreach $animal (@troupeau) {
    &{$animal."::fait"};
}
```


Le résultat sera :

```
un Boeuf fait mheuu !
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
un Mouton fait bêêê !
```

Super. Mais l'utilisation de références symboliques vers les sous-routines `fait` est un peu déplaisante. Nous comptons sur le mode `no strict subs` ce qui n'est certainement pas recommandé pour de gros programmes. Et pourquoi est-ce nécessaire ? Parce que le nom du paquetage semble inséparable du nom de la sous-routine que nous voulons appeler dans ce paquetage.

L'est-ce vraiment ?

9.1.2 Présentation de l'appel de méthodes via l'opérateur flèche

Pour l'instant, disons que `Class->method` appelle la sous-routine `method` du paquetage `Class`. (Ici, « `Class` » est utilisé dans le sens « catégorie » et non dans son sens « universitaire ».) Ce n'est pas tout à fait vrai mais allons-y pas à pas. Nous allons maintenant utiliser cela :

```
# Boeuf::fait, Cheval::fait, Mouton::fait comme au-dessus
Boeuf->fait;
Cheval->fait;
Mouton->fait;
```

À nouveau, le résultat sera :

```
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
```

Ce n'est pas encore super : même nombre de caractères, que des constantes, pas de variables. Mais maintenant, on peut séparer les choses :

```
$a = "Boeuf";
$a->fait; # appelle Boeuf->fait
```

Ah ! Maintenant que le nom du paquetage est séparé du nom de la sous-routine, on peut utiliser un nom de paquetage variable. Et cette fois, nous avons quelque chose qui marche même lorsque `use strict refs` est actif.

9.1.3 Et pour tout un troupeau

Prenons ce nouvel appel via l'opérateur flèche et appliquons-le dans l'exemple du troupeau :

```
sub Boeuf::fait {
    print "un Boeuf fait mheuu !\n";
}
sub Cheval::fait {
    print "un Cheval fait hiiii !\n";
}
sub Mouton::fait {
    print "un Mouton fait bêêê !\n"
}

@troupeau = qw(Boeuf Boeuf Cheval Mouton Mouton);
foreach $animal (@troupeau) {
    $animal->fait;
}
```

Ça y est ! Maintenant tous les animaux parlent et sans utiliser de référence symbolique.

Mais regardez tout ce code commun. Toutes les routines `fait` ont une structure similaire : un opérateur `print` et une chaîne qui contient un texte identique, exceptés deux mots. Ce serait intéressant de pouvoir factoriser les parties communes au cas où nous déciderions plus tard de changer `fait` en `dit` par exemple.

Il y a réellement moyen de le faire mais pour cela nous devons tout d'abord en savoir un peu plus sur ce que l'opérateur flèche peut faire pour nous.

9.1.4 Le paramètre implicite de l'appel de méthodes

L'appel :

```
Class->method(@args)
```

essaie d'appeler la subroutine `Class::method` de la manière suivante :

```
Class::method("Class", @args);
```

(Si la subroutine ne peut être trouvée, l'héritage intervient mais nous le verrons plus tard.) Cela signifie que nous récupérons le nom de la classe comme premier paramètre (le seul paramètre si aucun autre argument n'est fourni). Donc nous pouvons réécrire la subroutine `fait` du Mouton ainsi :

```
sub Mouton::fait {
    my $class = shift;
    print "un $class fait bêêê !\n";
}
```

Et, de manière similaire, pour les deux autres animaux :

```
sub Boeuf::fait {
    my $class = shift;
    print "un $class fait mheuu !\n";
}
sub Cheval::fait {
    my $class = shift;
    print "un $class fait hiiii !\n";
}
```

Dans chaque cas, `$class` aura la valeur appropriée pour la subroutine. Mais, encore une fois, nous avons des structures similaires. Pouvons-nous factoriser encore plus ? Oui, en appelant une autre méthode de la même classe.

9.1.5 Appel à une seconde méthode pour simplifier les choses

Créons donc une seconde méthode nommée `cri` qui sera appelée depuis `fait`. Cette méthode fournit un texte constant représentant le cri lui-même.

```
{ package Boeuf;
  sub cri { "mheuu" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Maintenant, lorsque nous appelons `Boeuf->fait`, `$class` vaut `Boeuf` dans `cri`. Cela permet de choisir la méthode `Boeuf->cri` qui retourne `mheuu`. Quelle différence voit-on dans la version correspondant au Cheval ?

```
{ package Cheval;
  sub cri{ "hiiii" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Seuls le nom du paquetage et le cri changent. Pouvons-nous donc partager la définition de `fait` entre le Boeuf et le Cheval ? Oui, grâce à l'héritage !

9.1.6 L'héritage

Nous allons définir un paquetage commun appelé `Animal` contenant la définition de `fait` :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Puis nous allons demander à chaque animal « d'hériter » de `Animal` :

```
{ package Boeuf;
  @ISA = qw(Animal);
  sub cri { "mheuu" }
}
```

Remarquez l'ajout du tableau `@ISA`. Nous y reviendrons dans un instant...

Qu'arrive-t-il maintenant lorsque nous appelons `Boeuf->fait` ?

Tout d'abord, Perl construit la liste d'arguments. Dans ce cas, c'est juste `Boeuf`. Puis Perl cherche la subroutine `Boeuf::fait`. Mais elle n'existe pas alors Perl regarde le tableau d'héritage `@Boeuf::ISA`. Il existe et contient le seul nom `Animal`.

Perl cherche alors `fait` dans `Animal`, c'est à dire `Animal::fait`. Comme elle existe, Perl appelle cette subroutine avec la liste d'arguments pré-calculée.

Dans la subroutine `Animal::fait`, `$class` vaut `Boeuf` (le premier et seul argument). Donc, lorsque nous arrivons à `$class->cri`, la recherche commence par `Boeuf->cri` qui est trouvé au premier essai sans passage par le tableau `@ISA`. Ça marche !

9.1.7 Quelques remarques au sujet de @ISA

La variable magique `@ISA` (qui se prononce - en anglais - « is a » et non « ice-uh ») déclare que `Boeuf` est un (« is a ») `Animal`. Notez bien que ce n'est pas une valeur mais bien un tableau ce qui permet, en de rares occasions, d'avoir plusieurs parents capables de fournir les méthodes manquantes.

Si `Animal` a lui aussi un tableau `@ISA`, alors il est utilisé aussi. Par défaut, la recherche est récursive, en profondeur d'abord et de gauche à droite dans chaque `@ISA` (pour changer ce comportement, voir *mro*). Classiquement, chaque `@ISA` ne contient qu'un seul élément (des éléments multiples impliquent un héritage multiple et donc de multiples casse-tête) ce qui définit un bel arbre d'héritage.

Lorsqu'on active `use strict`, on obtient un avertissement concernant `@ISA` puisque c'est une variable dont le nom ne contient pas explicitement un nom de paquetage et qui n'est pas déclarée comme une variable lexicale (via « my »). On ne peut pas en faire une variable lexicale (car elle doit appartenir au paquetage pour être accessible au mécanisme d'héritage). Voici donc plusieurs moyens de gérer cela :

Le moyen le plus simple est d'inclure explicitement le nom du paquetage :

```
@Boeuf::ISA = qw(Animal);
```

On peut aussi créer une variable de paquetage implicitement :

```
package Boeuf;
use vars qw(@ISA);
@ISA = qw(Animal);
```

Si vous préférez une méthode plus orientée objet, vous pouvez changer :

```
package Boeuf;
use Animal;
use vars qw(@ISA);
@ISA = qw(Animal);
```

en :

```
package Boeuf;
use base qw(Animal);
```

Ce qui est très compact.

9.1.8 Surcharge de méthodes

Ajoutons donc un mulot qu'on peut à peine entendre :

```
# Paquetage Animal comme au-dessus
{ package Mulot;
  @ISA = qw(Animal);
  sub cri { "fiiik" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n";
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}

Mulot->fait;
```

Le résultat sera :

```
un Mulot fait fiiik !
[mais vous pouvez à peine l'entendre !]
```

Ici, `Mulot` a sa propre routine `cri`, donc `Mulot->fait` n'appellera pas immédiatement `Animal->fait`. On appelle cela la surcharge (« overriding » en anglais). En fait, nous n'avons pas besoin du tout de dire qu'un `Mulot` est un `Animal` puisque toutes les méthodes nécessaires à `fait` sont définies par `Mulot`.

Mais maintenant nous avons dupliqué le code de `Animal->fait` et cela peut nous amener à des problèmes de maintenance. Alors, pouvons-nous l'éviter ? Pouvons-nous dire qu'un `Mulot` fait exactement comme un autre `Animal` mais en ajoutant un commentaire en plus ? Oui !

Tout d'abord, nous pouvons appeler directement la méthode `Animal::fait` :

```
# Paquetage Animal comme au-dessus
{ package Mulot;
  @ISA = qw(Animal);
  sub cri { "fiiik" }
  sub fait {
    my $class = shift;
    Animal::fait($class);
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Remarquez que nous sommes obligés d'inclure le paramètre `$class` (qui doit certainement valoir `Mulot`) comme premier paramètre de `Animal::fait` puisque nous n'utilisons plus l'opérateur flèche. Pourquoi ne plus l'utiliser ? Si nous appelons `Animal->fait` ici, le premier paramètre de la méthode sera "`Animal`" et non "`Mulot`" et lorsqu'on arrivera à l'appel de `cri`, nous n'aurons pas la bonne classe.

L'invocation directe de `Animal::fait` est tout autant problématique. Que se passe-t-il si la subroutine `Animal::fait` n'existe pas et est en fait héritée d'une classe mentionnée dans `@Animal::ISA` ? Puisque nous n'utilisons pas l'opérateur flèche, nous n'avons pas la moindre chance que cela fonctionne.

Notez aussi que le nom de la classe `Animal` est maintenant codée explicitement pour choisir la bonne subroutine. Ce sera un problème pour celui qui changera le tableau `@ISA` de `Mulot` sans remarquer que `Animal` est utilisé explicitement dans `fait`. Ce n'est donc pas la bonne méthode.

9.1.9 Effectuer la recherche à partir d'un point différent

Une meilleure solution consiste à demander à Perl de rechercher dans la chaîne d'héritage un cran plus haut :

```
# Animal comme au-dessus
{ package Mulot;
  # même @ISA et même &cri qu'au-dessus
  sub fait {
    my $class = shift;
    $class->Animal::fait;
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Ça marche. En utilisant cette syntaxe, nous commençons par chercher dans `Animal` pour trouver `fait` et nous utilisons toute la chaîne d'héritage de `Animal` si on ne la trouve pas tout de suite. Et comme le premier argument sera `$class`, la méthode `fait` trouvée pourra éventuellement appeler `Mulot::cri`.

Mais ce n'est pas la meilleure solution. Nous avons encore à coordonner le tableau `@ISA` et le nom du premier paquetage de recherche. Pire, si `Mulot` a plusieurs entrées dans son tableau `@ISA`, nous ne savons pas nécessairement laquelle définit réellement `fait`. Alors, y a-t-il une meilleure solution ?

9.1.10 Le SUPER moyen de faire des choses

En changeant la classe `Animal` par la classe `SUPER`, nous obtenons automatiquement une recherche dans toutes les `SUPER` classes (les classes listées dans `@ISA`) :

```
# Animal comme au dessus
{ package Mulot;
  # même @ISA et même &cri qu'au dessus
  sub dir {
    my $class = shift;
    $class->SUPER::fait;
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Donc, `SUPER::fait` signifie qu'il faut chercher la subroutine `fait` dans les paquetages listés par le tableau `@ISA` du paquetage courant (en commençant par le premier). Notez bien que la recherche *ne sera pas* faite dans le tableau `@ISA` de `$class`.

9.1.11 Où en sommes-nous ?

Jusqu'ici, nous avons vu la syntaxe d'appel des méthodes via la flèche :

```
Class->method(@args);
```

ou son équivalent :

```
$a = "Class";
$a->method(@args);
```

qui construit la liste d'argument :

```
("Class", @args)
```

et essaye d'appeler :

```
Class::method("Class", @args);
```

Si `Class::method` n'est pas trouvée, alors le tableau `@Class::ISA` est utilisé (récursivement) pour trouver un paquetage qui propose `method` puis la méthode est appelée.

En utilisant cette simple syntaxe, nous avons des méthodes de classes avec héritage (multiple), surcharge et extension. Et nous avons été capable de factoriser tout le code commun tout en fournissant un moyen propre de réutiliser l'implémentation avec des variantes. C'est ce que fournit le coeur de la programmation orientée objet mais les objets peuvent aussi fournir des données d'instances que nous n'avons pas encore vues.

9.1.12 Un cheval est un cheval bien sûr... Mais n'est-il que cela ?

Repartons donc du code de la classe `Animal` et de la classe `Cheval` :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
}
```

Cela permet d'appeler `Cheval->fait` qui est en fait `Animal::fait` qui, elle-même appelle en retour `Cheval::cri` pour obtenir le cri spécifique. Ce qui produit :

```
un Cheval fait hiiii !
```

Mais tous nos objets `Cheval` (`Chevaux` ;-)) doivent absolument être identiques. Si j'ajoute une subroutine, tous les chevaux la partagent automatiquement. C'est très bien pour faire des chevaux identiques mais, alors, comment faire pour distinguer les chevaux les uns des autres ? Par exemple, supposons que nous voulions donner un nom au premier cheval. Il nous faut un moyen de conserver son nom indépendamment des autres chevaux.

Nous pouvons le faire en introduisant une nouvelle notion appelée « instance ». Une « instance » est généralement créée par une classe. En Perl, n'importe quelle référence peut être une instance. Commençons donc par la plus simple des références qui peut stocker le nom d'un cheval : une référence sur un scalaire.

```
my $nom = "Mr. Ed";
my $parleur = \$nom;
```

Maintenant `$parleur` est une référence vers ce qui sera une donnée spécifique de l'instance (le nom). L'étape finale consiste à la transformer en une vraie instance grâce à l'opérateur appelé `bless` (*bénir* ou *consacrer* en anglais) :

```
bless $parleur, Cheval;
```

Cet opérateur associe le paquetage nommé `Cheval` à ce qui est pointé par la référence. À partir de ce moment, on peut dire que `$parleur` est une instance de `Cheval`. C'est à dire que c'est un `Cheval` spécifique. La référence en tant que telle n'est pas modifiée et on peut continuer à l'utiliser avec les opérateurs traditionnels de déréférencement.

9.1.13 Appel d'une méthode d'instance

L'appel de méthodes via l'opérateur flèche peut être utilisé sur des instances exactement comme on le fait avec un nom de paquetage (classe). Donc, cherchons le cri que `$parleur` fait :

```
my $bruit = $parleur->cri;
```

Pour appeler `cri`, Perl remarque tout d'abord que `$parleur` est une référence consacrée (via `bless()`) et donc une instance. Ensuite, il construit la liste des arguments qui, dans ce cas, n'est que `$parleur`. (Plus tard, nous verrons que les autres arguments suivent la variable d'instance exactement comme avec les classes.)

Maintenant la partie intéressante : Perl récupère la classe ayant consacré l'instance, dans notre cas `Cheval`, et l'utilise pour trouver la subroutine à appeler. Dans notre cas, `Cheval::cri` est trouvée directement (sans utiliser d'héritage) et cela nous amène à l'appel finale de la subroutine :

```
Cheval::cri($parleur)
```

Remarquez que le premier paramètre est bien l'instance et non le nom de la classe comme auparavant. Nous obtenons `hiiii` comme valeur de retour et cette valeur est stockée dans la variable `$bruit`.

Si `Cheval::cri` n'avait pas existé, nous aurions été obligé d'explorer `@Cheval::ISA` pour y rechercher cette subroutine dans l'une des super-classes exactement comme avec les méthodes de classes. La seule différence entre une méthode de classe et une méthode d'instance est le premier argument qui est soit un nom de classe (une chaîne) soit une instance (une référence consacrée).

9.1.14 Accès aux données d'instance

Puisque nous avons une instance comme premier paramètre, nous pouvons accéder aux données spécifiques de l'instance. Dans notre cas, ajoutons un moyen d'obtenir le nom :

```
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
  sub nom {
    my $self = shift;
    $$self;
  }
}
```

Maintenant, appelons cette méthode :

```
print $parleur->nom, " fait ", $parleur->cri, "\n";
```

Dans `Cheval::nom`, le tableau `@_` contient juste `$parleur` que `shift` stocke dans `$self`. (Il est classique de dépiler le premier paramètre dans une variable nommée `$self` pour les méthodes d'instance. Donc conservez cela tant que vous n'avez pas de bonnes raisons de faire autrement.) Puis, `$self` est déréférencé comme un scalaire pour obtenir `Mr. Ed`. Le résultat sera :

```
Mr. Ed fait hiiii
```

9.1.15 Comment fabriquer un cheval

Bien sûr, si nous construisons tous nos chevaux à la main, nous ferons des erreurs de temps en temps. Nous violons aussi l'un des principes de la programmation orientée objet puisque les « entrailles » d'un cheval sont visibles. C'est bien si nous sommes vétérinaire pas si nous sommes de simples propriétaires de chevaux. Laissons donc la classe `Cheval` fabriquer elle-même un nouveau cheval :

```
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
  sub nom {
    my $self = shift;
    $$self;
  }
  sub nomme {
    my $class = shift;
    my $nom = shift;
    bless \$nom, $class;
  }
}
```

Maintenant, grâce à la méthode `nomme`, nous pouvons créer un cheval :

```
my $parleur = Cheval->nomme("Mr. Ed");
```

Remarquez que nous sommes revenus à une méthode de classe donc les deux arguments de `Cheval::nomme` sont `Cheval` et `Mr. Ed`. L'opérateur `bless` en plus de consacrer `$nom` retourne une référence à `$nom` qui est parfaite comme valeur de retour. Et c'est comme cela qu'on construit un cheval.

Ici, nous avons appelé le constructeur `nomme` ce qui indique que l'argument de ce constructeur est le nom de ce `Cheval` particulier. Vous pouvez utiliser différents constructeurs avec différents noms pour avoir des moyens différents de « donner naissance » à un objet. En revanche, vous constaterez que de nombreuses personnes qui sont venues à Perl à partir de langages plus limités n'utilisent qu'un seul constructeur appelé `new` avec plusieurs façons d'interpréter ses arguments. Tous les styles sont corrects tant que vous documentez (et vous le ferez, n'est-ce pas ?) le moyen de donner naissance à votre objet.

9.1.16 Héritage de constructeur

Mais y a-t-il quelque chose de spécifique au Cheval dans cette méthode ? Non. Par conséquent, c'est la même chose pour construire n'importe quoi qui hérite d'un Animal. Plaçons donc cela dans Animal :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
  sub nom {
    my $self = shift;
    $$self;
  }
  sub nomme {
    my $class = shift;
    my $nom = shift;
    bless \$nom, $class;
  }
}
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
}
```

Bon. Mais que se passe-t-il si nous appelons fait depuis une instance ?

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->fait;
```

Nous obtenons le texte suivant :

```
un Cheval=SCALAR(0xaca42ac) fait hiiii !
```

Pourquoi ? Parce que la routine `Animal::fait` s'attend à recevoir un nom de classe comme premier paramètre et non une instance. Lorsqu'une instance est passée, nous nous retrouvons à utiliser une référence consacrée à un scalaire en tant que chaîne et nous obtenons ce que nous venons de voir.

9.1.17 Concevoir une méthode qui marche aussi bien avec des instances qu'avec des classes

Tout ce dont nous avons besoin c'est de détecter si l'appel se fait via une classe ou via une instance. Le moyen le plus simple est d'utiliser l'opérateur `ref`. Il retourne une chaîne (le nom de la classe) lorsqu'il est appliqué sur une référence consacrée et une chaîne vide lorsqu'il est appliqué à une chaîne (comme un nom de classe). Modifions donc la méthode `nom` pour prendre cela en compte :

```
sub nom {
  my $classouref = shift;
  ref $classouref
  ? $$classouref # c'est une instance, on retourne le nom
  : "un $classouref anonyme"; # c'est une classe, on retourne un nom générique
}
```

Ici, l'opérateur `?:` devient le moyen de choisir entre déréférencement ou chaîne. Maintenant nous pouvons utiliser notre méthode indifféremment avec une classe ou avec une instance. Notez que nous avons transformé le premier paramètre en `$classouref` pour indiquer ce qu'il contient :

```
my $parleur = Cheval->nomme("Mr. Ed");
print Cheval->nom, "\n"; # affiche "un Cheval anonyme\n"
print $parleur->nom, "\n"; # affiche "Mr. Ed\n"
```


Modifions fait pour utiliser nom :

```
sub fait {
    my $classouref = shift;
    print $classouref->nom, " fait ", $classouref->cri, "\n";
}
```

Et puisque cri fonctionne déjà que ce soit avec une instance ou une classe, nous avons fini !

9.1.18 Ajout de paramètres aux méthodes

Faisons manger nos animaux :

```
{ package Animal;
  sub nomme {
    my $class = shift;
    my $nom = shift;
    bless \$nom, $class;
  }
  sub nom {
    my $classouref = shift;
    ref $classouref
      ? $$classouref # c'est une instance, on retourne le nom
      : "un $classouref anonyme"; # c'est une classe, on retourne un nom générique
  }
  sub fait {
    my $classouref = shift;
    print $classouref->nom, " fait ", $classouref->cri, "\n";
  }
  sub mange {
    my $classouref = shift;
    my $nourriture = shift;
    print $classouref->nom, " mange $nourriture.\n";
  }
}
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
}
{ package Mouton;
  @ISA = qw(Animal);
  sub cri { "bêêê" }
}
```

Essayons ce code :

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->mange("du foin");
Mouton->mange("de l'herbe");
```

qui affiche :

```
Mr. Ed mange du foin.
un Mouton anonyme mange de l'herbe.
```

Une méthode d'instance avec des paramètres est appelé avec, comme paramètres, l'instance puis la liste des paramètres. Donc ici, le premier appel est comme :

```
Animal::mange($parleur, "du foin");
```

9.1.19 Des instances plus intéressantes

Comment faire pour qu'une instance possède plus de données ? Les instances les plus intéressantes sont constituées de plusieurs éléments qui peuvent être eux-mêmes des références ou même des objets. Le moyen le plus simple pour les stocker est souvent une table de hachage. Les clés de la table de hachage servent à nommer ces différents éléments (qu'on appelle souvent « variables d'instance » ou « variables membres ») et les valeurs attachées sont... les valeurs de ces éléments.

Mais comment transformer notre cheval en une table de hachage ? Rappelez-vous qu'un objet est une référence consacrée. Il est tout à fait possible d'utiliser une référence consacrée vers une table de hachage plutôt que vers un simple scalaire à partir du moment où chaque accès au contenu de cette référence l'utilise correctement.

Créons un mouton avec un nom et une couleur :

```
my $mauvais = bless { Nom => "Evil", Couleur => "noir" }, Mouton;
```

Ainsi `$mauvais->{Nom}` donne `Evil` et `$mauvais->{Couleur}` donne `Noir`. Mais `$mauvais->nom` doit donner le nom et cela ne marche plus car cette méthode attend une référence vers un simple scalaire. Ce n'est pas très grave car c'est simple à corriger :

```
## dans Animal
sub nom {
    my $classouref = shift;
    ref $classouref ?
        $classouref->{Nom} :
        "un $classouref anonyme";
}
```

Bien sûr, `nomme` construit encore un mouton avec une référence vers un scalaire. Corrigeons là aussi :

```
## dans Animal
sub nomme {
    my $class = shift;
    my $nom = shift;
    my $self = { Nom => $nom, Couleur => $class->couleur_defaut };
    bless $self, $class;
}
```

D'où vient ce `couleur_defaut` ? Eh bien, puisque `nomme` ne fournit que le nom, nous devons encore définir une couleur. Nous avons donc une couleur par défaut pour la classe. Pour un mouton, nous pouvons la définir à blanc :

```
## dans Mouton
sub couleur_defaut { "blanc" }
```

Et pour nous éviter de définir une couleur par défaut pour toutes les classes, nous allons définir aussi une méthode générale dans `Animal` qui servira de « couleur par défaut » par défaut :

```
## dans Animal
sub couleur_defaut { "marron" }
```

Comme `nom` et `nomme` étaient les seules méthodes qui utilisaient explicitement la « structure » des objets, toutes les autres méthodes restent inchangées et donc `fait` fonctionne encore comme avant.

9.1.20 Des chevaux de couleurs différentes

Des chevaux qui sont tous de la même couleur sont ennuyeux. Alors ajoutons une méthode ou deux afin de choisir la couleur.

```
## dans Animal
sub couleur {
    $_[0]->{Couleur}
}
sub set_couleur {
    $_[0]->{Couleur} = $_[1];
}
```

Remarquez un autre moyen d'utiliser les arguments : \$_[0] est utilisé directement plutôt que via un `shift`. (Cela économise un tout petit peu de temps pour quelque chose qui peut être invoqué fréquemment.) Maintenant, on peut choisir la couleur de Mr. Ed :

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->set_couleur("noir-et-blanc");
print $parleur->nom, " est de couleur ", $parleur->couleur, "\n";
```

qui donne :

```
Mr. Ed est de couleur noir-et-blanc
```

9.1.21 Résumé

Ainsi, maintenant nous avons des méthodes de classe, des constructeurs, des méthodes d'instances, des données d'instances, et également des accesseurs. Mais cela n'est que le début de ce que Perl peut offrir. Nous n'avons pas non plus commencé à parler des accesseurs qui fonctionnent à la fois en lecture et en écriture, des destructeurs, de la notation d'objets indirects, des sous-classes qui ajoutent des données d'instances, des données de classe, de la surcharge, des tests « isa » et « can » de la classe UNIVERSAL, et ainsi de suite. C'est couvert par le reste de la documentation de Perl. En espérant que cela vous permette de démarrer, vraiment.

9.2 VOIR AUSSI

Pour plus d'informations, voir *perlobj* (pour tous les petits détails au sujet des objets Perl, maintenant que vous avez vu les bases), *perltot* (le tutoriel pour ceux qui connaissent déjà les objets, la page de manuel *perltot* (qui traite les classes de données), *perlbot* (pour les trucs et astuces), et des livres tels que l'excellent *Object Oriented Perl* de Damian Conway. Citons quelques modules qui sont dignes d'intérêt `Class::Accessor`, `Class::Class`, `Class::Contract`, `Class::Data::Inheritable`, `Class::MethodMaker` et `Tie::SecureHash`

9.3 COPYRIGHT

Copyright (c) 1999, 2000 by Randal L. Schwartz and Stonehenge Consulting Services, Inc. Permission is hereby granted to distribute this document intact with the Perl distribution, and in accordance with the licenses of the Perl distribution; derived documents must include this copyright notice intact.

Portions of this text have been derived from Perl Training materials originally appearing in the *Packages, References, Objects, and Modules* course taught by instructors for Stonehenge Consulting Services, Inc. and used with permission.

Portions of this text have been derived from materials originally appearing in *Linux Magazine* and used with permission.

9.4 TRADUCTION

9.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

9.4.2 Traducteur

Paul Gaborit <Paul.Gaborit@enstimac.fr> avec la participation de Gérard Robin <robin.jag@free.fr>.

9.4.3 Relecture

Jean Forget <ponder.stibbons@wanadoo.fr>. Gérard Delafond.

Chapitre 10

perltoot

Tutoriel orienté objet de Tom.

10.1 DESCRIPTION

La programmation orientée objet se vend bien de nos jours. Certains managers préféreraient même s'arrêter de respirer plutôt que de se passer d'objets. Pourquoi cela ? Qu'est-ce qu'un objet a de si particulier ? Et plus simplement, qu'est-ce qu'un objet ?

Un objet n'est rien de plus qu'une manière de cacher des comportements complexes derrière un petit ensemble clair et simple à utiliser. (C'est ce que les professeurs appellent l'abstraction.) De brillant programmeurs qui n'ont rien à faire si ce n'est de réfléchir pendant des semaines entières sur des problèmes vraiment difficiles créent, pour les résoudre, de chouettes objets que des gens normaux peuvent utiliser (C'est ce que les professeurs appellent la réutilisation de programmes). Les utilisateurs (en fait, des programmeurs) peuvent jouer comme ils veulent avec ces objets tout prêts, mais ils n'ont pas à les ouvrir et à mettre la pagaille à l'intérieur. C'est comme un équipement coûteux : le contrat spécifie que la garantie disparaît si vous ouvrez le capot. Donc ne le faites pas.

Le coeur des objets est la classe, un espace de nommage protégé et un peu privé contenant données et fonctions. Une classe est un ensemble de routines relatives à un même problème. Vous pouvez aussi la considérer comme un type défini par l'utilisateur. Le mécanisme de paquetage de Perl, qui est aussi utilisé pour des modules traditionnels, sert pour les modules de classe. Les objets « vivent » dans une classe ce qui signifie qu'ils appartiennent à un paquetage.

La plupart du temps, la classe fournit à l'utilisateur de petits trucs. Ces trucs sont des objets. Ils savent à quelle classe ils appartiennent et comment se comporter. Les utilisateurs demandent quelque chose à la classe comme « donne-moi un objet ». Ou ils peuvent demander à l'un de ces objets de faire quelque chose. Demander à une classe de faire quelque chose pour vous consiste à appeler une *méthode de la classe*. Demander à un objet de faire quelque chose pour vous consiste à appeler une *méthode d'objet*. Demander soit à une classe (le cas usuel), soit à un objet (parfois) de vous retourner un objet consiste à appeler un *constructeur* qui est simplement une sorte de méthode.

Bon d'accord, mais en quoi un objet est-il différent de n'importe quel autre type de donnée en Perl ? Qu'est ce qu'un objet *réellement* ? Autrement dit, quel est son type de base ? La réponse à la première question est simple. Un objet n'a qu'une seule différence avec n'importe quel autre type de donnée en Perl : vous pouvez le déréférencer non seulement par une chaîne ou un nombre comme les tables de hachage ou les tableaux, mais aussi par des appels à des routines nommées. En un mot, les *méthodes*.

La réponse à la seconde question est que c'est une référence, mais pas n'importe quelle référence. Une référence qui a été *bénie* (blessed en anglais) par une classe particulière (lire: paquetage). Quel type de référence ? Bon, la réponse à cette question est un peu moins concrète, parce qu'en Perl, le concepteur de la classe peut utiliser n'importe quel type de référence. Ce peut être un scalaire, un tableau ou une table de hachage. Cela peut même être une référence à du code. Mais de par sa flexibilité inhérente, un objet est dans la plupart des cas une référence à une table de hachage.

10.2 Créer une classe

Avant de créer une classe, vous devez choisir son nom. Car le nom de la classe (du paquetage) détermine le nom du fichier qui la contiendra exactement comme pour les modules. Ensuite, cette classe devrait fournir un ou plusieurs moyens de créer des objets. Finalement, elle devrait proposer des mécanismes pour permettre à l'utilisateur de ces objets de les manipuler indirectement à distance.

Par exemple, créons un simple module pour la classe `Person`. Elle doit être stockée dans le fichier `Person.pm`. Si elle s'appelait `Happy::Person`, elle devrait être stockée dans le fichier `Happy/Person.pm` et son paquetage deviendrait `Happy::Person` au lieu de `Person`. (Sur un ordinateur personnel n'utilisant pas Unix ou Plan 9 mais quelque chose comme MacOS ou VMS, le séparateur de répertoires peut être différent, mais le principe reste le même.) Ne voyez aucune forme de relation entre les modules en vous basant sur le nom de leur répertoire. C'est simplement une facilité de regroupement et cela n'a aucun effet sur l'héritage, l'accessibilité des variables ou quoi que ce soit d'autres.

Pour ce module, nous n'utiliserons pas l'Exporter puisque nous construirons une classe bien élevée qui n'exportera rien du tout. Pour créer des objets, une classe doit avoir un *constructeur*. Un constructeur ne vous renvoie pas seulement un type normal de donnée, mais un objet tout neuf de cette classe. Cela est fait automatiquement par la fonction `bless()` dont le seul rôle est d'autoriser l'utilisation d'une référence en tant qu'objet. Rappelez-vous : être un objet ne signifie rien de plus que pouvoir appeler des méthodes à partir de soi-même.

Bien qu'un constructeur puisse porter le nom que l'on veut, la plupart des programmeurs Perl semble les appeler `new()`. Par contre `new()` n'est pas un mot réservé et une classe n'a aucune obligation de proposer une méthode portant ce nom. Quelques programmeurs utilisent aussi une fonction avec le même nom que la classe du constructeur.

10.2.1 Représentation des objets

Pour représenter une structure C ou une classe C++, le mécanisme le plus couramment utilisé en Perl, et de loin, est une table de hachage anonyme. Parce qu'une table de hachage peut contenir un nombre arbitraire de champs tous accessibles par un nom arbitrairement choisi.

Si vous voulez une émulation simple d'une structure, vous devriez écrire quelque chose comme :

```
$rec = {
    name => "Jason",
    age  => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

Si vous préférez, vous pouvez créer un peu de différence visuelle en mettant les clés en majuscules :

```
$rec = {
    NAME => "Jason",
    AGE  => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

Et vous utilisez `$rec->{NAME}` pour trouver "Jason" ou `@{ $rec->{PEERS} }` pour obtenir "Norbert", "Rhys", et "Phineas". (Avez-vous remarqué combien de programmeurs de 23 ans semblent s'appeler "Jason" ces temps-ci ? :-)

Ce modèle est parfois utilisé pour des classes bien que la possibilité offerte à n'importe qui à l'extérieur de la classe de modifier directement et impunément les données internes (les données de l'objet) ne soit pas considérée comme le summum d'une programmation de qualité. En général, un objet devrait apparaître comme un truc opaque auquel on accède via des *méthodes de l'objet*. Visuellement, les méthodes permettent de déréférencer une référence en utilisant un nom de fonction plutôt que des crochets ou des accolades.

10.2.2 L'interface d'une classe

Certains langages disposent d'une interface syntaxique formelle vers les méthodes d'une classe. Perl ne possède rien de tout cela. C'est à vous de lire la documentation de chaque classe. Si vous appelez une méthode non-définie pour un objet, Perl ne dira rien, mais votre programme engendrera une exception durant son exécution. De même, si vous appelez une méthode qui attend un nombre premier comme argument avec un nombre non-premier à la place, vous ne pouvez espérer que le compilateur le détecte. (En fait, vous pouvez toujours espérer, mais cela n'arrivera pas.)

Supposons que l'utilisateur de votre classe `Person` soit respectueux (quelqu'un qui a lu la documentation expliquant l'interface prescrite). Voici comment il devrait utiliser la classe `Person`:

```
use Person;
```

```

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him; # stockage dans un tableau pour plus tard

printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;

```

Comme vous pouvez le voir, l'utilisateur de la classe ne sait pas (ou au moins, n'a pas à faire attention) si l'objet a une implémentation particulière ou une autre. L'interface vers la classe et ses objets se fait exclusivement via des méthodes et c'est la seule chose avec laquelle l'utilisateur travaille.

10.2.3 Constructeurs et méthodes d'objet

Par contre, *quelqu'un* doit savoir ce qu'il y a dans l'objet. Ce quelqu'un c'est la classe. Elle implémente les méthodes que le programmeur utilise pour accéder à l'objet. Voici le manière d'implémenter la classe Person en utilisant l'idiome standard objet-référence-vers-table-de-hachage. Nous fabriquons un méthode de classe appelée new() en guise de constructeur et trois méthodes d'objet appelées name(), age() et peers() pour cacher l'accès à nos données d'objet dans notre table de hachage anonyme.

```

package Person;
use strict;

#####
## le constructeur d'objet (version simpliste) ##
#####
sub new {
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless($self); # voir ci-dessous
    return $self;
}

#####
## méthodes pour accéder aux données d'objet ##
## ##
## Avec des arguments, elles changent la(les) ##
## valeur(s) ##
## Sans, elles ne font que la(les) retrouver. ##
#####

sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

```

```

sub peers {
    my $self = shift;
    if (@_) { @{$self->{PEERS}} = @_ }
    return @{$self->{PEERS}};
}

1; # ainsi le 'require' ou le 'use' réussit

```

Nous avons créé trois méthodes pour accéder aux données de l'objet: `name()`, `age()` et `peers()`. Elles sont similaires. Si elles sont appelées avec un argument, elles stockent la nouvelle valeur du champ. Sans argument, elles retournent la valeur contenu dans le champ correspondant, c'est-à-dire la valeur indexée par cette clé dans la table de hachage.

10.2.4 En prévision du futur: de meilleurs constructeurs

Bien que pour l'instant vous ne sachiez peut-être pas du tout ce que c'est, il faut faire attention à l'héritage. (Vous pouvez ne pas vous en occuper pour l'instant et n'y revenir que plus tard.) Pour être sûr que tout cela fonctionne correctement, vous devez utiliser la fonction `bless()` sous sa forme à deux arguments. Le second argument est la classe par laquelle la référence doit être bénie (`blessed`). Si vous utilisez notre propre classe comme second argument par défaut plutôt que de retrouver la classe qui vous est passée, vous rendez votre constructeur inhéritable.

```

sub new {
    my $class = shift;
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}

```

C'est tout ce qu'il y a à savoir sur les constructeurs. Ces méthodes donnent vie aux objets et retournent à l'utilisateur un truc opaque qui sera utilisé plus tard pour appeler des méthodes.

10.2.5 Destructeurs

Toutes les histoires ont un début et une fin. Le début de l'histoire d'un objet est son constructeur qui est explicitement appelé à la création de l'objet. Le *destructeur* représente la fin de l'objet. Il est appelé implicitement lorsque l'objet disparaît. Tout code de nettoyage lié à l'objet doit être placé dans le destructeur qui (en Perl) doit s'appeler `DESTROY`.

Pourquoi les constructeurs peuvent-ils avoir des noms arbitraires et pas les destructeurs ? Parce qu'un constructeur est appelé explicitement ce qui n'est pas le cas du destructeur. La destruction se fait automatiquement via le système de ramasse-miettes (GC) de Perl qui est un GC à base de référence souvent rapide, mais parfois un peu indolent. Pour savoir quoi appeler, Perl impose que le destructeur s'appelle `DESTROY`. La notion de bon moment pour appeler le destructeur n'est pas très bien définie actuellement en Perl. C'est pourquoi vos destructeurs ne doivent pas dépendre du moment où ils sont appelés.

Pourquoi `DESTROY` est tout en majuscule ? C'est une convention en Perl que toutes les fonctions entièrement en majuscule peuvent être appelées automatiquement par Perl. Il en existe d'autres qui sont appelées implicitement comme `BEGIN`, `END`, `AUTOLOAD`, plus tous les méthodes utilisées par les objets `tied` (cravatés ?) et décrites dans *perltie*.

Dans les langages réellement orientés objets, l'utilisateur n'a pas à se préoccuper d'appeler le destructeur. Cela arrive automatiquement quand il faut. Dans les langages de bas niveau sans ramasse-miettes du tout, il n'y a aucun moyen d'automatiser cela. C'est donc au programmeur d'appeler explicitement le destructeur pour nettoyer la mémoire en croisant les doigts pour que ce soit le bon moment. Au contraire de C++, un destructeur est très rarement nécessaire en Perl et même quand il l'est, il n'est pas nécessaire de l'appeler explicitement. Dans le cas de notre classe `Person`, nous n'avons pas besoin de destructeur, car Perl s'occupe tout seul de pas mal de choses, comme la libération de la mémoire.

La seule situation où le ramasse-miettes de Perl échoue, c'est en présence de références circulaires comme celle-ci :

```
$this->{WHATEVER} = $this;
```

Dans ce cas, vous devez détruire manuellement la référence circulaire si vous voulez que votre programme n'ait pas de fuites mémoire. C'est l'idéal. Par contre, assurez-vous que lorsque votre programme se termine, tous les destructeurs de ces objets sont appelés. Ainsi vous êtes sûr qu'un objet sera détruit proprement, sauf dans le cas où votre programme ne se termine jamais. (Si vous faites tourner Perl à l'intérieur d'une autre application, cette passe complète du ramasse-miettes peut arriver plus souvent – par exemple, à la fin de chaque fil d'exécution.)

10.2.6 Autres méthodes d'objets

Les méthodes dont nous avons parlé jusqu'à présent sont soit des constructeurs, soit de simples méthodes d'accès aux données stockées dans l'objet. Cela ressemble un peu aux données membres des objets du monde C++, sauf que dans ce cas les "étrangers" n'y accèdent pas comme des données. À la place, ils doivent y accéder indirectement via des méthodes. Règle importante : en Perl, l'accès aux données d'un objet ne devrait se faire *que* par l'intermédiaire des méthodes.

Perl n'impose aucune restriction sur qui utilise quelles méthodes. La distinction publique/privée n'est pas syntaxique. C'est une convention. (Sauf si vous utilisez le module `Alias` décrit plus bas dans `Les données membres comme des variables`.) De temps en temps, vous verrez des méthodes dont le nom commence ou finit par un ou deux caractères de soulignement. C'est une convention signifiant que ces méthodes sont privées et utilisables uniquement par cette classe, et éventuellement des classes proches ou ses sous-classes. Mais Perl ne fait pas respecter cette distinction. C'est au programmeur de bien se comporter.

Il n'y a aucune raison de limiter l'usage des méthodes à l'accès aux données de l'objet. Une méthode peut faire tout ce qu'on veut. Le point clé est de savoir si elle est appelée comme méthode de classe ou comme méthode d'objet. Supposons que vous vouliez écrire une méthode objet qui fasse plus que donner l'accès en lecture et/ou écriture à un champ particulier :

```
sub exclaim {
    my $self = shift;
    return sprintf "Hello, Je suis %s, j'ai %d ans, je travaille avec %s",
        $self->{NAME}, $self->{AGE}, join(", ", @{$self->{PEERS}});
}
```

ou quelque chose comme ça :

```
sub joyeux_anniversaire {
    my $self = shift;
    return ++$self->{AGE};
}
```

Certains diront qu'il vaut mieux les écrire comme ça :

```
sub exclaim {
    my $self = shift;
    return sprintf "Hello, Je suis %s, j'ai %d ans, je travaille avec %s",
        $self->name, $self->age, join(", ", $self->peers);
}

sub joyeux_anniversaire {
    my $self = shift;
    return $self->age( $self->age() + 1 );
}
```

Mais comme ces méthodes sont toutes exécutées dans la classe elle-même, ce n'est pas critique. Le choix est une histoire de compromis. L'utilisation directe de la table de hachage est plus rapide (d'un ordre de grandeur pour être précis). Mais l'usage des méthodes (c'est à dire l'interface externe) protège non seulement l'utilisateur de votre classe, mais aussi vous-même des éventuels changements dans la représentation interne de vos données objet.

10.3 Données de classe

Que dire des données de classe, les données communes à tous les objets de la classe ? À quoi peuvent-elles vous servir ? Supposons que dans votre classe `Person`, vous vouliez garder une trace du nombre total de personnes. Comment implémenter cela ?

Vous *pourriez* en faire une variable globale appelée `$Person::Census`. Mais la seule justification valable pour agir ainsi serait de donner au gens la possibilité d'accéder directement à votre donnée de classe. Ils pourraient utiliser `$Person::Census` pour faire ce qu'ils veulent. Peut-être est-ce ce qu'il vous faut. Vous pourriez même en faire une variable exportée. Pour être exportable, une variable doit être globale (au paquetage). Si c'était un module traditionnel plutôt qu'un module orienté objet, c'est comme cela qu'il faudrait faire.

Bien qu'étant l'approche utilisée dans la plupart des modules traditionnels, cette manière de faire est considérée comme mauvaise dans une approche objet. Dans un module objet, vous devriez pouvoir séparer l'interface de l'implémentation. Il faut donc proposer des méthodes de classe pour accéder aux données de la classe comme le font les méthodes objets pour accéder aux données des objets.

Donc, vous *pourriez* garder `$Census` comme une variable globale en espérant que les autres respecteront le contrat de modularité en n'accédant pas directement à son implémentation. Vous pourriez même être plus rusé (voire retors) et faire de `$Census` un objet « tied » comme décrit dans *perlite* et donc intercepter tous les accès.

Le plus souvent, il vous suffira de déclarer votre donnée de classe avec une portée lexicale locale au fichier. Pour cela, il vous suffit de mettre la ligne suivante au début de votre fichier :

```
my $Census = 0;
```

Normalement, la portée d'une variable `my()` se termine en même temps que le bloc dans lequel elle est déclarée (dans notre cas ce serait l'ensemble du fichier requis (par `require()`) ou utilisé (par `use()`)), mais en fait le mécanisme de gestion des variables lexicales implémenté par Perl garantit que la variable ne sera pas désallouée et restera accessible à toutes les fonctions ayant la même portée. Par contre, cela ne fonctionne pas avec les variables globales auxquelles on donne une valeur temporaire via `local()`.

Indépendamment de la méthode choisie (`$Census` comme variable globale du paquetage ou avec une portée lexicale locale au fichier), vous devrez modifier le constructeur `$Person::new()` de la manière suivante :

```
sub new {
    my $class = shift;
    my $self = {};
    $Census++;
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Ceci étant fait, nous avons maintenant besoin d'un destructeur afin de décrémenter `$Census` lorsqu'un objet `Person` est détruit. Voici ce destructeur :

```
sub DESTROY { --$Census }
```

Remarquez qu'il n'y a pas de mémoire à désallouer dans le destructeur ! C'est Perl qui s'en occupe pour vous tout seul. Pour faire tout cela vous pouvez aussi utiliser le module `Class::Data::Inheritable` disponible sur CPAN.

10.3.1 Accès aux données de classe

Il s'avère que ce n'est pas vraiment une bonne chose que de manipuler les données de classe. Une bonne règle est : *vous ne devriez jamais référencer directement des données de classe directement dans une méthode objet*. Car, sinon, vous ne construisez pas une classe héritable. L'objet doit être le point de rendez-vous pour toutes les opérations, et en particulier depuis les méthodes objets. Les données globales (les données de classe) peuvent être dans le mauvais paquetage du point de vue des classes héritées. En Perl, les méthodes s'exécutent dans le contexte de la classe où elles sont définies et *pas* dans celui de l'objet qui les a appelées. Par conséquent, la visibilité des variables globales d'un paquetage dans les méthodes n'est pas liée à l'héritage.

Ok, supposons qu'une autre classe emprunte (en fait hérite de) la méthode `DESTROY` telle qu'elle est définie précédemment. Lorsque ses objets sont détruits, c'est la variable originale `$Census` qui est modifiée et non pas celle qui est dans l'espace de noms du paquetage de la nouvelle classe. La plupart du temps, cela ne correspond pas à ce que vous vouliez.

Bon, voici comment y remédier. Nous allons stocker une référence à la variable dans la valeur associée à la clé `"_CENSUS"` de la table de hachage de l'objet. Pourquoi un caractère de soulignement au début ? Principalement parce que cela évoque une notion magique à un programmeur C. C'est en fait un moyen mnémotechnique pour nous souvenir que ce champ est spécial et qu'il ne doit pas être utilisé comme une donnée publique telle que `NAME`, `AGE` ou `PEERS`. (En raison de l'utilisation du `pragma strict`, dans les versions antérieures à la 5.004, nous devons entourer de guillemets le nom du champ.)

```

sub new {
    my $class = shift;
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    # données "privées"
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}

```

10.3.2 Méthodes de débogage

Une classe propose souvent un mécanisme de débogage. Par exemple, vous pourriez vouloir voir quand les objets sont créés et détruits. Pour cela, il vous faut ajouter un variable de débogage de portée lexicale limitée au fichier. Nous utiliserons aussi le module standard Carp pour émettre nos avertissements (warnings) et nos messages d'erreur. Ainsi ces messages s'afficheront avec le nom et le numéro de la ligne du fichier de l'utilisateur plutôt qu'avec ceux de notre fichier. Si nous les voulons de notre point de vue, il nous suffit d'utiliser respectivement die() et warn() plutôt que croak() et carp()

```

use Carp;
my $Debugging = 0;

```

Ajoutons maintenant une nouvelle méthode de classe pour accéder à cette variable.

```

sub debug {
    my $class = shift;
    if (ref $class) { confess "Class method called as object method" }
    unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
    $Debugging = shift;
}

```

Modifions DESTROY pour afficher un petit quelque chose lorsqu'un objet disparaît:

```

sub DESTROY {
    my $self = shift;
    if ($Debugging) { carp "Destroying $self " . $self->name }
    -- ${ $self->{"_CENSUS"} };
}

```

Il est concevable d'avoir un mécanisme de débogage par objet. Afin de pouvoir utiliser les deux appels suivants :

```

Person->debug(1); # toute la classe
$him->debug(1);  # juste un objet

```

Nous avons donc besoin de rendre "bimodale" notre méthode debug afin qu'elle fonctionne à la fois sur la classe *et* sur les objets. Modifions donc debug() et DESTROY comme suit :

```
sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self)) {
        $self->{"_DEBUG"} = $level;        # juste moi-même
    } else {
        $Debugging        = $level;        # toute la classe
    }
}

sub DESTROY {
    my $self = shift;
    if ($Debugging || $self->{"_DEBUG"}) {
        carp "Destroying $self " . $self->name;
    }
    -- ${ $self->{"_CENSUS"} };
}
```

Que se passe-t-il si une classe dérivée (que nous appellerons Employee) hérite de ces méthodes depuis la classe de base Person ? Eh bien, Employee->debug(), lorsqu'elle est appelée en tant que méthode de classe, modifie \$Person::Debugging et non \$Employee::Debugging.

10.3.3 Destructeurs de classes

Le destructeur d'objet traite la disparition de chaque objet. Mais parfois, vous devez faire un peu de nettoyage lorsque toute la classe disparaît. Ce qui n'arrive actuellement qu'à la fin du programme. Pour faire un *destructeur de classe*, créez une fonction END dans le paquetage de la classe. Elle fonctionne exactement comme la fonction END des modules traditionnels. Ce qui signifie qu'elle est appelée lorsque votre programme se termine, à moins qu'il n'effectue un 'exec' ou qu'il meurt sur un signal non capté. Par exemple :

```
sub END {
    if ($Debugging) {
        print "All persons are going away now.\n";
    }
}
```

Quand le programme se termine, tous les destructeurs de classes (les fonctions END) sont exécutés dans l'ordre inverse de leur chargement (LIFO).

10.3.4 La documentation de l'interface

Jusqu'ici nous n'avons exhibé que *l'implémentation* de la classe Person. Son *interface* doit être sa documentation. Habituellement cela signifie d'ajouter du pod ("plain old documentation") dans le même fichier. Dans le cas de notre exemple Person, nous devons placer la documentation suivante quelque part dans le fichier Person.pm. Bien que cela ressemble à du code, ce n'en est pas. C'est de la documentation intégrée utilisable par des programmes comme pod2man, pod2html ou pod2text. Le compilateur Perl ne tient pas compte des parties pod. À l'inverse, les traducteurs pod ne tiennent pas compte des parties de code. Voici un exemple de pod décrivant notre interface :

```
=head1 NAME

Person - classe pour implémenter des gens

=head1 SYNOPSIS

use Person;
```

```
#####
# méthodes de classe #
#####
$obj = Person->new;
$count = Person->population;

#####
# méthodes d'accès aux données d'objets #
#####

### accès en lecture ###
$who = $obj->name;
$years = $obj->age;
@pals = $obj->peers;

### accès en écriture ###
$obj->name("Jason");
$obj->age(23);
$obj->peers( "Norbert", "Rhys", "Phineas" );

#####
# autres méthodes d'objets #
#####

$phrase = $obj->exclaim;
$obj->joyeux_anniversaire;

=head1 DESCRIPTION
```

La classe Person permet de blah, blah, blah...

C'est donc tout ce qui concerne l'interface et non pas l'implémentation. Un programmeur qui ouvre le module pour jouer avec toutes les astuces d'implémentation qui sont cachées derrière le contrat d'interface rompt la garantie et vous n'avez pas à vous préoccuper de son sort.

10.4 Agrégation

Supposons que, plus tard, vous vouliez modifier la classe pour avoir une meilleure gestion des noms. Par exemple, en gérant le prénom, le nom de famille, mais aussi les surnoms et les titres. Si les utilisateurs de votre classe Person y accèdent proprement via l'interface documentée, vous pouvez alors changer sans risque l'implémentation sous-jacente. S'ils ne l'ont pas fait, ils ont tout perdu, mais c'est leur faute puisqu'en rompant le contrat, ils perdent la garantie.

Nous allons donc créer une nouvelle classe appelée Fullname. À quoi ressemblera cette classe Fullname ? Pour pouvoir répondre, nous devons d'abord examiner comment nous comptons l'utiliser. Par exemple:

```
$him = Person->new();
$him->fullname->title("St");
$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
printf "His normal name is %s\n", $him->name;
printf "But his real name is %s\n", $him->fullname->as_string;
```

Ok. Changeons Person::new() pour qu'il accepte un champ fullname :

```

sub new {
    my $class = shift;
    my $self = {};
    $self->{FULLNAME} = Fullname->new();
    $self->{AGE}       = undef;
    $self->{PEERS}    = [];
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub fullname {
    my $self = shift;
    return $self->{FULLNAME};
}

```

Puis, pour accepter le vieux code, définissons `Person::name()` de la manière suivante :

```

sub name {
    my $self = shift;
    return $self->{FULLNAME}->nickname(@_)
        || $self->{FULLNAME}->christian(@_);
}

```

Voici maintenant la classe `Fullname`. Là encore, nous utilisons une table de hachage pour stocker les données associées à des méthodes avec le nom qui va bien pour y accéder :

```

package Fullname;
use strict;

sub new {
    my $class = shift;
    my $self = {
        TITLE       => undef,
        CHRISTIAN   => undef,
        SURNAME     => undef,
        NICK        => undef,
    };
    bless ($self, $class);
    return $self;
}

sub christian {
    my $self = shift;
    if (@_) { $self->{CHRISTIAN} = shift }
    return $self->{CHRISTIAN};
}

sub surname {
    my $self = shift;
    if (@_) { $self->{SURNAME} = shift }
    return $self->{SURNAME};
}

sub nickname {
    my $self = shift;
    if (@_) { $self->{NICK} = shift }
    return $self->{NICK};
}

```

```

sub title {
    my $self = shift;
    if (@_) { $self->{TITLE} = shift }
    return $self->{TITLE};
}

sub as_string {
    my $self = shift;
    my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
    if ($self->{TITLE}) {
        $name = $self->{TITLE} . " " . $name;
    }
    return $name;
}

1;

```

Pour finir, voici un programme de test :

```

#!/usr/bin/perl -w
use strict;
use Person;
sub END { show_census() }

sub show_census () {
    printf "Current population: %d\n", Person->population;
}

Person->debug(1);

show_census();

my $him = Person->new();

$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
$him->fullname->title("St");
$him->age(1);

printf "%s is really %s.\n", $him->name, $him->fullname->as_string;
printf "%s's age: %d.\n", $him->name, $him->age;
$him->happy_birthday;
printf "%s's age: %d.\n", $him->name, $him->age;

show_census();

```

10.5 Héritage

Tous les systèmes de programmation orienté objets incluent sous une forme ou une autre la notion d'héritage. L'héritage permet à une classe d'englober une autre classe afin d'éviter de ré-écrire la même chose plusieurs fois. C'est en rapport avec la réutilisation logicielle et donc avec la paresse qui est, comme chacun sait, la vertu principale d'un programmeur. (Le mécanisme d'import/export des modules traditionnels est aussi une forme de réutilisation de code, mais beaucoup plus simple que le vrai héritage qu'on trouve dans les modules objets.)

Parfois la syntaxe de l'héritage est construite au coeur du langage, mais ce n'est pas toujours le cas. Perl n'a aucune syntaxe spéciale pour spécifier la classe (ou les classes) dont on hérite. À la place, tout est fait purement sémantiquement. Chaque paquetage peut contenir une variable appelée @ISA qui pilote l'héritage (des méthodes). Si vous essayez d'appeler une

méthode d'un objet ou d'une classe et que cette méthode n'est pas trouvée dans le paquetage de l'objet, Perl trouve dans @ISA le nom d'autres paquetages où chercher la méthode manquante.

Comme les variables spéciales de paquetages reconnues par Exporter (telles que @EXPORT, @EXPORT_OK, @EXPORT_FAIL, %EXPORT_TAGS, et \$VERSION), le tableau @ISA *doit* être une variable de portée globale au paquetage et non une variable de portée lexicale limitée au fichier et créée par my(). La plupart des classes n'ont qu'un seul nom dans leur tableau @ISA. C'est ce que nous appellerons de l'"Héritage Simple" ou HS pour faire court.

Examinons la classe suivante :

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

Ce n'est pas grand chose, hein ? Tout ce que ça fait c'est de charger une autre classe et d'indiquer qu'on hérite des méthodes de cette autre classe, si nécessaire. Nous n'avons spécifié aucune de ses propres méthodes. Nous obtenons donc un Employee qui se comporte exactement comme une Person.

Une telle classe vide est appelée une "sous-classe vide de test". C'est une classe qui ne fait rien si ce n'est d'hériter d'une classe de base. Si la classe originale est correctement conçue, alors la nouvelle classe dérivée peut être utilisée en remplacement de celle d'origine. Cela veut dire que vous pouvez écrire un programme comme celui-ci :

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

Par correctement conçue, nous entendons toujours utiliser bless() dans sa forme à deux arguments, éviter l'accès direct aux données globales et ne rien exporter. En regardant la fonction Person::new() que nous avons définie précédemment, vous pourrez vérifier tout cela. Quelques données du paquetage sont utilisées dans le constructeur, mais leur référence est stockée dans l'objet lui-même et toutes les autres méthodes accèdent à ces données via ces références. Ça doit donc fonctionner.

Qu'entendons-nous par la fonction Person::new(), – n'est-ce pas une méthode ? En principe, oui. Une méthode est simplement une fonction qui attend comme premier paramètre soit un nom de classe (de paquetage), soit un objet (une référence bénie). Person::new() est la fonction que les deux méthodes Person->new() et Employee->new() appellent. Bien qu'un appel à une méthode ressemble à un appel de fonction, il faut bien comprendre que ce n'est pas exactement la même chose. Si vous les considérez comme identiques, vous vous retrouverez très rapidement avec des programmes complètement boggués. Tout d'abord, les conventions d'appel sous-jacentes sont différentes: l'appel à une méthode ajoute implicitement un argument supplémentaire. Ensuite, les appels de fonctions n'utilisent pas les mécanismes d'héritage mis en oeuvre pour les méthodes.

Appel de méthode	Appel de fonction résultant
-----	-----
Person->new()	Person::new("Person")
Employee->new()	Person::new("Employee")

Donc, n'utilisez pas un appel de fonction à la place d'un appel de méthode.

Si un Employee est juste une Person, cela n'est pas très utile. Ajoutons donc quelques méthodes. Nous allons ajouter à nos Employee des données d'objet pour leur salaire (salary), leur identification (ID number) et leur date d'embauche (start date).

Si vous êtes fatigué de créer ces méthodes d'accès toutes bâties sur le même principe, ne désespérez pas. Plus tard, nous décrirons différentes techniques pour automatiser cela.

```
sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}
```

```

sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}

```

10.5.1 Polymorphisme

Qu'arrive-t-il lorsqu'une classe dérivée et sa classe de base définissent toutes deux la même méthode ? La méthode utilisée est la version de la classe dérivée. Par exemple, supposons que nous voulions que la méthode `peers()` agisse différemment lorsque nous l'appelons pour un `Employee`. Au lieu de renvoyer simplement la liste des `peers`, nous aimerions obtenir une autre chaîne. De telle sorte que :

```

$empl->peers("Peter", "Paul", "Mary");
printf "His peers are: %s\n", join(" ", $empl->peers);

```

produira :

```

His peers are: PEON=PETER, PEON=PAUL, PEON=MARY

```

Pour obtenir ce résultat, il faut ajouter la définition suivante dans le fichier `Employee.pm` :

```

sub peers {
    my $self = shift;
    if (@_) { @{$self->{PEERS}} = @_ }
    return map { "PEON=\U$_" } @{$self->{PEERS}};
}

```

Nous venons d'illustrer le concept de *polymorphisme*. Nous avons réutilisé la forme et le comportement d'un objet existant, puis nous l'avons modifié pour l'adapter à nos besoins. C'est une forme de Paresse. (Être polymorphe c'est aussi ce qui vous arrive quand un magicien décide que vous seriez mieux sous la forme d'une grenouille.)

Supposons maintenant que nous voulions un appel de méthode qui déclenche à la fois la version de la classe dérivée (aussi appelée sous-classe) et la version de la classe de base (aussi appelée super-classe). En pratique, c'est le cas des constructeurs et des destructeurs et probablement celui de la méthode `debug()` dont nous avons parlé précédemment.

Pour cela, ajoutons ce qui suit dans `Employee.pm` :

```

use Carp;
my $Debugging = 0;

sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)" unless @_ == 1;
    my $level = shift;
    if (ref($self)) {
        $self->{"_DEBUG"} = $level;
    } else {
        $Debugging = $level;          # toute la classe
    }
    Person::debug($self, $Debugging); # à ne pas faire !
}

```

Comme vous pouvez le constater nous appelons directement la fonction `debug()` du paquetage `Person`. Mais cela est beaucoup trop spécifique pour être une bonne conception. En effet, que se passe-t-il si `Person` n'a pas de fonction `debug()`, mais en hérite d'ailleurs ? Il aurait été préférable de dire :


```
Person->debug($Debugging);
```

Mais même cela est encore de trop bas niveau. Il vaudrait mieux dire :

```
$self->Person::debug($Debugging);
```

C'est une drôle de manière pour demander de commencer la recherche de la méthode `debug()` à partir de la classe `Person`. Cette stratégie est plus souvent utilisée pour des méthodes d'objets que pour des méthodes de classe.

Cela laisse encore à désirer. Nous avons codé en dur le nom de notre super-classe. Ce qui n'est pas bon, en particulier si nous changeons la classe dont on hérite ou si on en ajoute d'autres. Heureusement, la pseudo-classe `SUPER` a été créée pour nous sauver.

```
$self->SUPER::debug($Debugging);
```

Comme ça, la recherche s'effectue dans tous les classes du tableau `@ISA`. Cela n'a un sens *que* lors de l'appel d'une méthode. N'essayez pas d'utiliser `SUPER` dans un autre contexte parce qu'elle n'existe que dans le cas d'appel à des méthodes redéfinies. Notez aussi que `SUPER` se réfère à la super-classe du package courant et *non* à celle de `$self`.

Tout cela est devenu un peu compliqué maintenant. Avons-nous tout fait comme il fallait ? Comme précédemment, pour vérifier que notre classe est correctement conçue, nous allons utiliser une sous-classe vide de test. Puisque nous avons déjà une classe `Employee` que nous voulons tester, notre classe de test sera dérivée de `Employee`. En voici une :

```
package Boss;
use Employee;          # :-)
@ISA = qw(Employee);
```

Et voici le programme de test :

```
#!/usr/bin/perl -w
use strict;
use Boss;
Boss->debug(1);

my $boss = Boss->new();

$boss->fullname->title("Don");
$boss->fullname->surname("Pichon Alvarez");
$boss->fullname->christian("Federico Jesus");
$boss->fullname->nickname("Fred");

$boss->age(47);
$boss->peers("Frank", "Felipe", "Faust");

printf "%s is age %d.\n", $boss->fullname->as_string, $boss->age;
printf "His peers are: %s\n", join(", ", $boss->peers);
```

Son exécution nous montre que tout marche bien. Si vous voulez afficher votre objet sous une forme lisible comme le fait la commande `'x'` du déboguer, vous pouvez utiliser le module `Data::Dumper` disponible au CPAN :

```
use Data::Dumper;
print "Here's the boss:\n";
print Dumper($boss);
```

Ce qui devrait vous afficher quelque chose comme :

```

Here's the boss:
$VAR1 = bless( {
  _CENSUS => \1,
  FULLNAME => bless( {
    TITLE => 'Don',
    SURNAME => 'Pichon Alvarez',
    NICK => 'Fred',
    CHRISTIAN => 'Federico Jesus'
  }, 'Fullname' ),
  AGE => 47,
  PEERS => [
    'Frank',
    'Felipe',
    'Faust'
  ]
}, 'Boss' );

```

Heu... Il manque quelque chose. Où sont les champs `salary`, `start_date` et `ID` ? Nous ne leur avons jamais donné de valeur, même pas `undef`, donc ils n'apparaissent pas dans les clés de la table de hachage. La classe `Employee` ne définit pas sa propre méthode `new()` et la méthode `new()` de la classe `Person` ne sait rien des `Employee` (elle ne le doit pas : une bonne conception orientée objet suppose qu'une sous-classe a le droit de connaître les super-classes dont elle hérite, mais jamais le contraire). Créons donc `Employee::new()` comme suit :

```

sub new {
  my $class = shift;
  my $self = $class->SUPER::new();
  $self->{SALARY}      = undef;
  $self->{ID}          = undef;
  $self->{START_DATE} = undef;
  bless ($self, $class);      # reconsecrate
  return $self;
}

```

À présent, si vous affichez un objet `Employee` ou `Boss`, vous verrez ces nouveaux champs.

10.5.2 Héritage multiple

Bon, au risque d'ennuyer les gourous OO et de perturber les débutants, il est temps d'avouer que le système objet de Perl propose la notion très controversée d'héritage multiple (ou HM pour faire court). Cela signifie qu'au lieu d'hériter d'une classe qui elle-même peut hériter d'une autre classe et ainsi de suite, vous pouvez hériter directement de plusieurs classes parentes. Il est vrai que le HM peut rendre les choses confuses, même si en Perl cela l'est un peu moins qu'avec des langages OO douteux comme le C++.

La manière dont cela fonctionne est vraiment très simple : il suffit de mettre plus d'un nom de paquetage dans le tableau `@ISA`. Lorsque Perl cherche une méthode pour votre objet, il regarde dans chacun de ces paquetages dans l'ordre. C'est une recherche récursive en profondeur d'abord (par défaut ; voir *mro* pour utiliser d'autres ordres de recherche de méthodes). Supposons un ensemble de tableaux `@ISA` comme :

```

@First::ISA = qw( Alpha );
@Second::ISA = qw( Beta );
@Third::ISA = qw( First Second );

```

Si vous avez un objet de la classe `Third` :

```

my $ob = Third->new();
$ob->spin();

```

Comment allons-nous trouver la méthode `find()` (ou une méthode `new()`) ? Puisque la recherche s'effectue en profondeur d'abord, les classes seront explorées dans l'ordre suivant : `Third`, `First`, `Alpha`, `Second` et enfin `Beta`.

En pratique, très peu de modules connus font usage de l'HM. La plupart préfèrent choisir l'inclusion d'une classe dans une autre plutôt que l'HM. C'est pourquoi notre objet `Person` contient un objet `Fullname`. Cela ne veut pas dire qu'il en est un.

Par contre, il y a un domaine où l'HM de Perl est très répandu : lorsqu'une classe emprunte des méthodes à une autre classe. C'est assez commun spécialement pour quelques classes "pas très objet" comme `Exporter`, `DynaLoader`, `AutoLoader` et `SelfLoader`. Ces classes ne proposent pas de constructeurs. Elles n'existent que pour vous permettre d'hériter de leurs méthodes. (Le choix de l'héritage plutôt que de l'importation traditionnelle n'est pas entièrement clair.)

Par exemple, voici le tableau `@ISA` du module `POSIX` :

```
package POSIX;
@ISA = qw(Exporter DynaLoader);
```

Ce module `POSIX` n'est pas vraiment un module objet pas plus qu'un `Exporter` ou un `DynaLoader`. Ces derniers ne font que prêter leur comportement à `POSIX`.

Pourquoi n'utilise-t-on pas plus l'HM pour les méthodes objet ? La raison principale réside dans les effets de bord complexes. Par exemple, votre graphe (ce n'est pas obligatoirement un arbre) d'héritage peut converger vers la même classe de base. Bien que Perl empêche l'héritage récursif, avoir des parents qui se réfèrent à un même ancêtre n'est pas interdit aussi incestueux que cela paraisse. Que se passe-t-il si dans notre classe `Third` nous voulons que sa méthode `new()` appelle aussi les constructeurs de ses deux classes parentes ? La notation `SUPER` ne trouvera que la méthode de la première classe. D'autre part, qu'arrive-t-il si les classes `Alpha` et `Beta` ont toutes les deux un ancêtre commun ? disons `Nought`. Si vous parcourez l'arbre d'héritage afin d'appeler les méthodes cachées, vous finirez par appeler deux fois la méthode `Nought::new()`, ce qui est sûrement une mauvaise chose.

10.5.3 UNIVERSAL: la racine de tous les objets

Ne serait-ce pas pratique si tous les objets héritaient d'une même classe de base ? Ainsi, on pourrait avoir des méthodes communes à tous les objets sans avoir à ajouter explicitement cette classe dans chaque tableau `@ISA`. En fait, cela existe déjà. Vous ne le voyez pas, mais Perl suppose tacitement qu'il y a un élément supplémentaire à la fin de `@ISA` : la classe `UNIVERSAL`. Dans la version 5.003, il n'y avait aucune méthode prédéfinie dans cette classe, mais vous pouviez y mettre tout ce que vouliez.

Maintenant, depuis la version 5.004 (ou quelques versions subversives comme la 5.003_08), `UNIVERSAL` contient déjà des méthodes. Elles sont incluses dans votre binaire Perl et ne consomment donc aucun temps supplémentaire en chargement. Ces méthodes prédéfinies comprennent `isa()`, `can()` et `VERSION()`. `isa()` vous permet de savoir si un objet (respectivement une classe) "est" un autre objet (respectivement une autre classe) sans que vous ayez à parcourir vous-même la hiérarchie d'héritage :

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

La méthode `can()`, appelée à partir d'un objet ou d'une classe, vous indique si la chaîne passée en argument est le nom d'une méthode appellable de cette classe. En fait, elle vous renvoie même une référence vers la fonction de cette méthode :

```
$his_print_method = $obj->can('as_string');
```

Finalement, la méthode `VERSION` vérifie que la classe (ou la classe de l'objet) possède une variable globale appelée `$VERSION` dont la valeur est suffisamment grande. Exemple :

```
Some_Module->VERSION(3.0);
$his_vers = $ob->VERSION();
```

En général, vous n'avez pas à appeler `VERSION` vous-même. (Souvenez-vous qu'un nom de fonction tout en majuscule est une convention Perl pour indiquer que cette fonction est parfois appelée automatiquement par Perl.) Dans le cas de `VERSION`, cela arrive quand vous dites :

```
use Some_Module 3.0;
```

Si vous voulez ajouter un contrôle de version à votre classe `Person`, ajoutez juste dans `Person.pm` :

```
our $VERSION = '1.1';
```

Et vous pouvez donc dire dans `Employee.pm` :

```
use Person 1.1;
```

Et il sera sûr que vous avez au moins ce numéro de version. Ce n'est pas la même chose que d'exiger un numéro de version précis. Actuellement, aucun mécanisme n'existe pour installer simultanément de multiples versions d'un même module. Lamentable.

10.5.4 Approfondissements de quelques détails de UNIVERSAL

Il est tout à fait possible (même si c'est le plus souvent contre-indiqué) d'ajouter de noms de packages dans `@UNIVERSAL::ISA`. Ces packages seront alors implicitement hérités pour toutes les classes, comme l'est `UNIVERSAL`. En revanche, ni `UNIVERSAL` ni aucun de ses parents ne sont des classes de base explicites des objets. Essayons de clarifier cela en posant ce qui suit :

```
@UNIVERSAL::ISA = ('REALLYUNIVERSAL');

package REALLYUNIVERSAL;
sub special_method { return "123" }

package Foo;
sub normal_method { return "321" }
```

L'appel à `Foo->special_method()` retournera "123" mais les appels `Foo->isa('REALLYUNIVERSAL')` et `Foo->isa('UNIVERSAL')` retourneront la valeur fausse.

Même si votre classe utilise un ordre de résolution non-standard pour retrouver ses méthodes tel l'ordre C3 (voir *mro*), la résolution des méthodes via `UNIVERSAL / @UNIVERSAL::ISA` restera celle par défaut (en profondeur d'abord et de gauche à droite) et n'arrivera que si votre résolution via C3 n'aboutit pas.

Tout ce qu'on vient d'exposer ci-dessus est sûrement plus compréhensible en étudiant ce qui se passe réellement lors de la recherche d'une méthode et qui ressemble approximativement au pseudo-code suivant :

```
get_mro(class) {
    # recurses down the @ISA's starting at class,
    # builds a single linear array of all
    # classes to search in the appropriate order.
    # The method resolution order (mro) to use
    # for the ordering is whichever mro "class"
    # has set on it (either default (depth first
    # l-to-r) or C3 ordering).
    # The first entry in the list is the class
    # itself.
}

find_method(class, methname) {
    foreach $class (get_mro(class)) {
        if($class->has_method(methname)) {
            return ref_to($class->$methname);
        }
    }
    foreach $class (get_mro(UNIVERSAL)) {
        if($class->has_method(methname)) {
            return ref_to($class->$methname);
        }
    }
    return undef;
}
```

En revanche, le code qui implémente `UNIVERSAL::isa` ne cherche pas dans `UNIVERSAL` lui-même. Il explore uniquement le véritable `@ISA` du package.

10.6 Autre représentation d'objet

Rien n'oblige à implémenter des objets sous la forme de référence à une table de hachage. Un objet peut-être n'importe quel type de référence tant que cette référence a été bénie (*blessed*). Il peut donc être une référence à un scalaire, à un tableau ou à du code.

Un scalaire peut suffire si l'objet n'a qu'une valeur à stocker. Un tableau fonctionne dans la plupart des cas, mais rend l'héritage plus délicat puisqu'il vous faut créer de nouveaux indices pour les classes dérivées.

10.6.1 Des objets sous forme de tableaux

Si l'utilisateur de votre classe respecte le contrat et colle à l'interface déclarée, vous pouvez changer l'interface sous-jacente quand vous le voulez. Voici une autre implémentation qui respecte la même spécification d'interface. Cette fois, pour représenter l'objet, nous utilisons une référence à un tableau plutôt qu'une référence à une table de hachage.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

#####
## le constructeur de Person (version tableau) ##
#####
sub new {
    my $self = [];
    $self->[$NAME] = undef; # this is unnecessary
    $self->[$AGE] = undef; # as is this
    $self->[$PEERS] = []; # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}

sub peers {
    my $self = shift;
    if (@_) { @{$self->[$PEERS]} = @_ }
    return @{$self->[$PEERS]};
}

1; # so the require or use succeeds
```

Vous pourriez penser que l'accès au tableau est plus rapide que l'accès à la table de hachage, mais ils sont en fait comparables. Le tableau est *un tout petit peu* plus rapide, mais pas plus de 10 ou 15%, même si vous remplacez les variables comme \$AGE par des nombres comme 1. La plus grande différence entre les deux approches est l'utilisation de la mémoire. La représentation par table de hachage prend plus de mémoire que la représentation par tableau parce qu'il faut allouer de la mémoire pour stocker les clés d'accès en plus des valeurs. Par contre, ce n'est pas vraiment mauvais puisque, depuis la version 5.004, le mémoire n'est allouée qu'une seule fois pour une clé donnée indépendamment du nombre de tables de hachage qui utilisent cette clé. Il est même prévu qu'un jour ces différences disparaissent, quand des représentations sous-jacentes efficaces seront inventées.

Ceci étant, le petit gain en vitesse (ainsi que celui en mémoire) est suffisant pour inciter des programmeurs à choisir la représentation par tableau pour des classes simples. Il reste encore un petit problème d'extensibilité. Par exemple, quand vous aurez besoin de créer des sous-classes, vous constaterez que les tables de hachage marchent mieux.

10.6.2 Des objets sous forme de fermeture (closure)

Utiliser une référence à du code pour représenter un objet ouvre des perspectives fascinantes. Vous pouvez créer une nouvelle fonction anonyme (une fermeture) qui est la seule à pouvoir accéder aux données de l'objet. Parce que vous mettez les données dans une table de hachage anonyme dont la portée lexicale est limitée à la fermeture que vous créez, bénissez et renvoyez comme objet. Les méthodes de cet objet appellent la fermeture comme n'importe quelle subroutine normale en lui passant le champ qu'elles veulent modifier. (Oui, le double appel de fonction est lent, mais si vous voulez de la vitesse, vous ne devriez pas utiliser d'objets du tout, non ? :-)

L'utilisation devrait rester comme précédemment :

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

mais l'implémentation est radicalement (et peut-être même sublimement) différente :

```
package Person;

sub new {
    my $class = shift;
    my $self = {
        NAME => undef,
        AGE  => undef,
        PEERS => [],
    };
    my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name { &{ $_[0] }("NAME", @_[ 1 .. $#_ ] ) }
sub age  { &{ $_[0] }("AGE", @_[ 1 .. $#_ ] ) }
sub peers { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Le concept de fermeture provient de la programmation fonctionnelle et, à ceux qui sont résolument attachés à la programmation procédurale ou à la programmation orientée objet, l'objet caché derrière une référence à du code doit vraisemblablement rester mystérieux. L'objet créé et retourné par la méthode `new()` n'est plus une référence vers des données comme nous l'avons vu auparavant. C'est une référence à du code anonyme qui a accès à sa propre version des données de l'objet (liaison lexicale et instanciation) qui est stockée dans la variable privée `$self`. Bien que ce soit la même fonction (NDT: le même code) à chaque fois, il contient une version différente de `$self`.

Quand une méthode comme `$him->name("Jason")` est appelée, son "zéroième" argument implicite est l'objet appelant – exactement comme pour tous les appels de méthodes. Mais dans notre cas, c'est une référence à notre code (quelque chose comme un pointeur sur fonction en C++, mais avec une liaison vers des variables lexicales). À part l'appeler, on ne peut pas faire grand chose d'une référence vers du code. C'est donc exactement ce que nous faisons en disant `&{ $_[0] }`. C'est juste un appel à une fonction et non pas un appel de méthode. Le premier argument est la chaîne "NAME", tous les autres arguments sont ceux qui ont été passés à la méthode.

Lors de l'exécution de la fermeture créée par `new()`, la référence `$self` à la table de hachage devient soudainement visible. La fermeture retrouve son premier argument ("NAME" dans ce cas puisque c'est ce que lui passe la méthode `name()`) et l'utilise comme clé d'accès à cette table de hachage privée qui est cachée dans sa propre version de `$self`.

Rien ne permet à quiconque d'accéder à ces données cachées en dehors de l'exécution de cette méthode. Ou presque rien : vous *pourriez* utiliser le débogueur en allant pas à pas jusque dans le code de la méthode et voir les données, mais en dehors de cela aucune chance d'y accéder.

Bon, si tout ça n'intéresse pas les adeptes de Scheme, je ne vois ce qui pourrait les intéresser. La transposition de ces techniques en C++, Java ou tout autre langage de conception statique est laissée comme exercice futile à leurs aficionados.

Via la fonction `caller()`, vous pouvez même ajouter un peu plus de confidentialité en contraignant la fermeture à n'accepter que les appels provenant de son propre paquetage. Cela devrait sans aucun doute satisfaire les plus exigeants...

Vous avez eu ici de quoi satisfaire votre orgueil (la troisième vertu principale du programmeur). Plus sérieusement, l'orgueil est tout simplement la fierté de l'artisan qui vient d'écrire un bout de code bien conçu.

10.7 AUTOLOAD: les méthodes mandataires

L'auto-chargement (ou `autoload`) est un moyen d'intercepter l'appel à une méthode non définie. Une subroutine d'auto-chargement peut soit créer une nouvelle fonction au vol, soit en charger une depuis le disque, soit l'évaluer elle-même. Cette stratégie de définition au vol est ce qu'on appelle l'auto-chargement.

Mais ce n'est qu'une approche possible. Dans une autre approche, c'est la méthode d'auto-chargement qui fournit elle-même le service demandée. Utilisée de cette manière, on peut alors considérer cette méthode d'auto-chargement comme une méthode "proxy".

Quand Perl essaie d'appeler une fonction non définie dans un paquetage particulier et que cette fonction n'est pas définie, il cherche alors dans le même paquetage une fonction appelée `AUTOLOAD`. Si elle existe, elle est appelée avec les mêmes arguments que la fonction originale. Le nom complet de la fonction dans la variable `$AUTOLOAD` du paquetage. Une fois appelée, la fonction peut faire tout ce qu'elle veut et, entre autres, définir une nouvelle fonction avec le bon nom, puis faire une sorte de `goto` vers elle en s'effaçant de la pile d'appel.

Quel rapport avec les objets ? Après tout, nous ne parlons que de fonctions, pas de méthodes. En fait, puisqu'une méthode n'est qu'une fonction avec un argument supplémentaire et quelques sémantiques sympathiques permettant de la retrouver, nous pouvons aussi utiliser l'auto-chargement pour les méthodes. Perl ne commence la recherche de méthodes par auto-chargement que lorsqu'il a fini l'exploration via `@ISA`. Certains programmeurs ont même défini une méthode `UNIVERSAL::AUTOLOAD` pour intercepter tous les appels à des méthodes non définies pour n'importe quelle sorte d'objets.

10.7.1 Méthodes auto-chargées d'accès aux données

Vous êtes peut-être resté un peu sceptique devant la duplication de code que nous avons exposé tout d'abord dans la classe `Person`, puis dans la classe `Employee`. Chaque méthode d'accès aux données de la table de hachage est virtuellement identique. Cela devrait chatouiller votre plus grande vertu de programmeur: l'impatience. Mais votre paresse l'a emporté et vous n'avez rien fait. Heureusement, les méthodes proxy sont le remède à ce problème.

Au lieu d'écrire une nouvelle fonction à chaque fois que nous avons besoin d'un nouveau champ, nous allons utiliser le mécanisme d'auto-chargement pour générer (en fait, simuler) les méthodes au vol. Pour vérifier que nous accédons à un membre valide, nous le rechercherons dans le champ `_permitted` (qui, en anglais, se prononce "under-permitted"). Ce champ est une référence à une table de hachage de portée lexicale limitée au fichier (comme une variable `C` statique d'un fichier) contenant les champs autorisés et appelée `%fields`. Pourquoi le caractère de soulignement ? Pour le même raison que celui de `_CENSUS` : c'est un indicateur qui signifie "à usage interne seulement".

Voici à quoi ressemblent le code d'initialisation et le constructeur de la classe si nous suivons cette approche :

```
package Person;
use Carp;
our $AUTOLOAD; # it's a package global

my %fields = (
    name      => undef,
    age       => undef,
    peers     => undef,
);
```

```

sub new {
    my $class = shift;
    my $self = {
        _permitted => \%fields,
        %fields,
    };
    bless $self, $class;
    return $self;
}

```

Si nous voulons spécifier des valeurs par défaut pour nos champs, nous pouvons remplacer les `undef` dans la table de hachage `%fields`.

Avez-vous remarqué comment nous stockons la référence à nos données de classe dans l'objet lui-même ? Souvenez-vous qu'il est fondamental d'accéder aux données de classe à travers l'objet lui-même plutôt que d'utiliser directement `%fields` dans les méthodes. Sinon vous ne pourrez pas convenablement hériter.

La vraie magie réside dans notre méthode proxy qui gèrera tous les appels à des méthodes non définies pour des objets de la classe `Person` (ou des sous-classes de `Person`). Elle doit s'appeler `AUTOLOAD`. Encore une fois, son nom est tout en majuscule parce qu'elle est implicitement appelée par Perl et non pas directement par l'utilisateur.

```

sub AUTOLOAD {
    my $self = shift;
    my $type = ref($self)
        or croak "$self is not an object";

    my $name = $AUTOLOAD;
    $name =~ s/.*://; # strip fully-qualified portion

    unless (exists $self->{_permitted}->{$name} ) {
        croak "Can't access '$name' field in class $type";
    }

    if (@_) {
        return $self->{$name} = shift;
    } else {
        return $self->{$name};
    }
}

```

Assez chouette, non ? La seule chose à faire pour ajouter de nouveaux champs est de modifier `%fields`. Il n'y a aucune nouvelle fonction à écrire.

J'aurais même pu supprimer complètement le champ `_permitted`, mais je voulais illustrer comment stocker une référence à une donnée de classe dans un objet de manière à ne pas accéder à cette donnée directement dans les méthodes.

10.7.2 Méthodes d'accès aux données auto-chargées et héritées

Qu'en est-il de l'héritage ? Pouvons-nous définir la classe `Employee` de la même manière ? Oui, si nous faisons un peu attention.

Voici comment faire :

```

package Employee;
use Person;
use strict;
our @ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);

```



```

sub new {
    my $class = shift;
    my $self = $class->SUPER::new();
    my($element);
    foreach $element (keys %fields) {
        $self->{_permitted}->{$element} = $fields{$element};
    }
    @{$self}{keys %fields} = values %fields;
    return $self;
}

```

Une fois cela fait, nous n'avons même pas à définir une fonction AUTOLOAD dans le paquetage Employee puisque la version fournie par Person via l'héritage fonctionne très bien.

10.8 Méta-outils classiques

Même si l'approche par les méthodes proxy est plus pratique pour fabriquer des classes ressemblant à des structures que l'approche fastidieuse nécessitant le codage de chacune des fonctions d'accès, elle laisse encore un peu à désirer. Par exemple, vous devez gérer les appels erronés que vous ne voulez pas capter via votre proxy. Il faut aussi faire attention lors de l'héritage comme nous l'avons montré précédemment.

Les programmeurs Perl ont répondu aux besoins en créant différentes classes de construction de classes. Ces méta-classes sont des classes qui créent d'autres classes. Deux d'entre elles méritent notre attention : Class::Struct et Alias. Celles-ci ainsi que d'autres classes apparentées peuvent être récupérées dans le répertoire modules du CPAN.

10.8.1 Class::Struct

La plus vieille de toutes est Class::Struct. En fait, sa syntaxe et son interface étaient esquissées alors même que perl5 n'existait pas encore. Elle fournit le moyen de "déclarer" une classe d'objets dont le type de chaque champ est spécifié. La fonction permettant de faire cela s'appelle (rien de surprenant) struct(). Puisque les structures ou les enregistrements ne sont pas des types de base de Perl, à chaque fois que vous voulez créer une classe pour fournir un objet de type structure, vous devez vous-même définir la méthode new() ainsi que les méthodes d'accès aux données pour chacun des champs. Très rapidement, vous trouverez cela enquiquinant (pour ne pas dire autre chose). La fonction Class::Struct::struct() vous soulagera de cette tâche ennuyeuse.

Voici un simple exemple d'utilisation :

```

use Class::Struct qw(struct);
use Jobbie; # défini par l'utilisateur; voir plus bas

struct 'Fred' => {
    one      => '$',
    many     => '@',
    profession => 'Jobbie', # n'appelle pas de Jobbie->new()
};

$obj = Fred->new(profession => Jobbie->new());
$obj->one("hmmmm");

$obj->many(0, "here");
$obj->many(1, "you");
$obj->many(2, "go");
print "Just set: ", $obj->many(2), "\n";

$obj->profession->salary(10_000);

```

Les types des champs dans la structure peuvent être déclarés comme des types de base de Perl ou comme des types utilisateurs (classes). Les types utilisateurs seront initialisés par l'appel de la méthode new() de la classe.

Attention au fait que l'objet Jobbie n'est pas créé automatiquement par la méthode new() de la class Fred. Vous devez donc spécifier un objet Jobbie lorsque vous créez un instance de Fred.

Voici un exemple réel d'utilisation de la génération de structure. Supposons que vous vouliez modifier le fonctionnement des fonctions Perl gethostbyname() et gethostbyaddr() pour qu'elles renvoient des objets qui fonctionnent comme des structures C. Nous ne voulons pas d'OO, ici. Nous voulons seulement que ces objets se comportent comme des structures au sens C du terme.

```

use Socket;
use Net::hostent;
$h = gethostbyname("perl.com"); # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);

```

Voici comment faire en utilisant le module Class::Struct. Le point crucial est cet appel :

```

struct 'Net::hostent' => [          # notez le crochet
    name      => '$',
    aliases   => '@',
    addrtype  => '$',
    'length'  => '$',
    addr_list => '@',
];

```

qui crée les méthodes d'objets avec les bons noms et types. Il crée même la méthode new() pour vous.

Vous auriez pu implémenter votre objet de la manière suivante :

```

struct 'Net::hostent' => {          # notez l'accolade
    name      => '$',
    aliases   => '@',
    addrtype  => '$',
    'length'  => '$',
    addr_list => '@',
};

```

Class::Struct aurait alors utilisé une table de hachage anonyme comme type d'objet plutôt qu'un tableau anonyme. Le tableau est plus rapide et plus petit, mais la table de hachage fonctionne mieux si éventuellement vous voulez faire de l'héritage. Puisque dans le cas de cet objet de type structure nous ne prévoyons pas d'héritage, nous opterons cette fois-ci plutôt pour la rapidité et le gain de place que pour une meilleure flexibilité.

Voici l'implémentation complète :

```

package Net::hostent;
use strict;

BEGIN {
    use Exporter    ();
    our @EXPORT     = qw(gethostbyname gethostbyaddr gethost);
    our @EXPORT_OK  = qw(
        $h_name      @h_aliases
        $h_addrtype  $h_length
        @h_addr_list $h_addr
    );
    our %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
our @EXPORT_OK;

# Class::Struct interdit l'usage de @ISA
sub import { goto &Exporter::import }

use Class::Struct qw(struct);
struct 'Net::hostent' => [
    name      => '$',
    aliases   => '@',
    addrtype  => '$',
    'length'  => '$',
    addr_list => '@',
];

```

```

sub addr { shift->addr_list->[0] }

sub populate (@) {
    return unless @_;
    my $hob = new(); # Class::Struct made this!
    $h_name     = $hob->[0]           = $_[0];
    @h_aliases  = @{ $hob->[1] } = split ' ', $_[1];
    $h_addrtype = $hob->[2]           = $_[2];
    $h_length   = $hob->[3]           = $_[3];
    $h_addr     =                    $_[4];
    @h_addr_list = @{ $hob->[4] } = @_[ 4 .. $#_ ];
    return $hob;
}

sub gethostbyname ($) { populate(CORE::gethostbyname(shift)) }

sub gethostbyaddr ($;$) {
    my ($addr, $addrtype);
    $addr = shift;
    require Socket unless @_;
    $addrtype = @_ ? shift : Socket::AF_INET();
    populate(CORE::gethostbyaddr($addr, $addrtype))
}

sub gethost($) {
    if ($_[0] =~ /^(\d+(?:\.\d+(?:\.\d+(?:\.\d+)?))?)?$/) {
        require Socket;
        &gethostbyaddr(Socket::inet_aton(shift));
    } else {
        &gethostbyname;
    }
}

1;

```

Otre la création dynamique de classes, nous n'avons qu'effleuré certains concepts comme la redéfinition des fonctions de base, l'import/export de bits, le prototypage de fonctions, les raccourcis d'appels de fonctions via `&whatever` et le remplacement de fonction par `goto &whatever`. Ils ont tous un sens du point de vue des modules traditionnels, mais comme vous avez pu le constater, vous pouvez aussi les utiliser dans un module objet.

Dans la version 5.004 de Perl, vous pouvez trouver d'autres modules objet qui redéfinissent les fonctions de base avec des structures : `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime`, `User::grent` et `User::pwent`. Le composant final du nom de tous ces modules est entièrement en minuscules qui, par convention, sont réservés aux pragma du compilateur parce qu'ils modifient la compilation et les fonctions internes. Ils proposent en outre les noms de type qu'un programmeur C attend.

10.8.2 Les données membres comme des variables

Si vous utilisez des objets C++, vous êtes habitué à accéder aux données membres d'un objet par simples variables lorsque vous êtes dans une méthode. Le module `Alias` offre entre autres cette fonctionnalité, ainsi que la possibilité d'avoir des méthodes privées que les objets peuvent appeler, mais qui ne peuvent l'être depuis l'extérieur de la classe.

Voici un exemple de classe `Person` créée en utilisant le module `Alias`. Quand vous mettez à jour ces instances des variables, vous mettez à jour automatiquement les champs correspondants de la table de hachage. Pratique, non ?

```

package Person;

# C'est la même chose qu'avant...
sub new {
    my $class = shift;
    my $self = {

```

```

        NAME => undef,
        AGE  => undef,
        PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
our ($NAME $AGE $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return    $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return    $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return    @PEERS;
}

sub exclaim {
    my $self = attr shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $NAME, $AGE, join(", ", @PEERS);
}

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}

```

La déclaration `our` est nécessaire parce que le module `Alias` bidouille les variables globales du paquetage qui ont le même nom que les champs. Pour pouvoir utiliser des variables globales avec `use strict`, vous devez les déclarer au préalable. Ces variables globales du paquetage sont localisées dans le bloc englobant l'appel à `attr()` exactement comme si vous aviez utilisé `local()`. Par contre, cela signifie qu'elles sont considérées comme des variables globales avec des valeurs temporaires comme avec tout autre `local()`.

Il serait joli de combiner `Alias` avec quelque chose comme `Class::Struct` ou `Class::MethodMaker`.

10.9 NOTES

10.9.1 Terminologie objet

Dans la littérature OO, un grand nombre de mots différents sont utilisés pour nommer quelques concepts. Si vous n'êtes pas déjà un programmeur objet vous n'avez pas à vous en préoccuper. Dans le cas contraire, vous aimeriez sûrement savoir à quoi correspondent ces concepts en Perl.

Par exemple, un objet est souvent appelé une *instance* d'une classe et les méthodes objet *méthodes d'instance*. Les champs spécifiques de chaque objet sont souvent appelés *données d'instance* ou *attributs d'objet*. Tandis que les champs communs à tous les membres de la classe sont nommés *donnée de classe*, *attributs de classe* ou <données membres statiques>.

Classe de base, *classe générique* et *super classe* recouvrent tous la même notion. *Classe dérivée*, *classe spécifique* et *sous-classe* décrivent aussi la même chose.

Les programmeurs C++ ont des *méthodes statiques* et des *méthodes virtuelles*, Perl ne propose que les *méthodes de classe* et les *méthodes d'objet*. En fait, Perl n'a que des méthodes. Qu'une méthode s'applique à une classe ou à un objet ne se fait qu'à l'usage. Vous pouvez accidentellement appeler une méthode de classe (une méthode qui attend une chaîne comme argument implicite) comme une méthode d'objet (une méthode qui attend une référence comme argument implicite) ou vice-versa.

Du point de vue C++, toutes les méthodes en Perl sont virtuelles. C'est la raison pour laquelle il n'y a jamais de vérification des arguments par rapport aux prototypes des fonctions comme cela peut se faire pour les fonctions prédéfinies ou définies par l'utilisateur.

Puisque une classe... (NDT: je n'ai pas compris ce paragraphe !) Because a class is itself something of an object, Perl's classes can be taken as describing both a "class as meta-object" (also called *object factory*) philosophy and the "class as type definition" (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

10.10 VOIR AUSSI

Vous trouverez sans aucun doute de plus amples informations en lisant les documentations suivantes : *perlmod*, *perlref*, *perlobj*, *perlbot*, *perltie* et *overload*.

perlboot est un tutoriel simple et facile pour la programmation orientée objet.

perltoc donne plus de détails sur les données de classe.

Quelques modules particulièrement intéressants sont `Class::Accessor`, `Class::Class`, `Class::Contract`, `Class::Data::Inheritable`, `Class::MethodMaker` et `Tie::SecureHash`.

10.11 AUTEURS ET COPYRIGHT

Copyright (c) 1997, 1998 Tom Christiansen Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code en précisant l'origine serait de bonne courtoisie mais n'est pas obligatoire.

10.11.1 Remerciements

Merci à Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton et à beaucoup d'autres pour leurs remarques pertinentes.

10.12 TRADUCTION

La traduction française est distribuée avec les mêmes droits que sa version originale (voir ci-dessus).

10.12.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

10.12.2 Traducteur

Traduction et mise à jour : Paul Gaborit (Paul.Gaborit at enstimac.fr).

10.12.3 Relecture

Philippe de Visme (philippe@devisme.com).

Chapitre 11

perltooc

Le tutoriel de Tom pour les données de classe OO en Perl

11.1 DESCRIPTION

Lors de la conception d'une classe objet, vous ressentez parfois le besoin de partager des choses communes à tous les objets de cette classe. De tels *attributs de classe* ressemblent beaucoup à des variables globales à la classe tout entière, mais à la différence des variables globales à un programme, les attributs de classe n'ont de signification que pour la classe elle-même.

Voici quelques exemples pratiques d'utilisation d'attributs de classe :

- mémoriser le nombre d'objets créés, ou savoir combien existent encore ;
- extraire le nom ou le descripteur de fichier d'un fichier de log utilisé par une méthode de debugging ;
- accéder à des données communes, par exemple le total des opérations gérées par l'ensemble des ATM un jour donné ;
- accéder au dernier objet créé pour une classe, ou bien à l'objet le plus utilisé, ou encore extraire une liste d'objets.

A la différence d'une variable globale, les attributs de classe ne sont pas directement accessibles. En revanche, on peut connaître - et éventuellement modifier - leur état ou leur valeur, et ce uniquement au moyen des *méthodes de classe*. Ces méthodes d'accès aux attributs de classe ressemblent, dans l'esprit et dans leur réalisation, aux méthodes d'accès utilisées pour manipuler l'état des attributs d'instance d'un objet de la classe. Elles fournissent un coupe-feu efficace entre l'interface et l'implémentation.

Il est possible d'autoriser l'accès aux attributs de classe soit par le nom de classe, soit par une instance quelconque de cette classe. Si `$an_object` est un objet du type `Some_Class`, et si la méthode `&Some_Class::population_count` accède aux attributs de classe, alors les deux appels suivants sont possibles, et quasiment équivalents.

```
Some_Class->population_count()  
$an_object->population_count()
```

Maintenant, où va-t-on ranger la variable à laquelle accède cette méthode ? A la différence de langages plus restrictifs tels que C++, dans lesquels ces variables portent le nom de données membres statiques, Perl ne fournit pas de mécanisme syntaxique pour déclarer les attributs de classe, pas plus qu'il ne fournit de mécanisme syntaxique pour déclarer les attributs d'instance. Perl fournit au développeur un large ensemble de caractéristiques puissantes mais flexibles dont il peut se servir pour tel ou tel besoin particulier en fonction de la situation.

Une classe Perl est typiquement implémentée dans un module. Un module contient deux ensembles de caractéristiques complémentaires : un paquetage pour l'interface avec le reste du monde, et un espace lexical pour l'intimité. On peut utiliser l'un ou l'autre de ces mécanismes pour implémenter des attributs de classe. Ce qui signifie que vous devez choisir de loger vos attributs de classe soit dans les variables de paquetage, soit dans les variables lexicales.

Mais ce ne sont pas les seules décisions à prendre. Si l'on choisit d'utiliser les variables de paquetage, il faudra également décider si les méthodes d'accès aux attributs seront sensibles ou insensibles à l'héritage. Si à l'inverse le choix se porte sur les variables lexicales, il faudra décider si l'on étend leur visibilité au fichier entier, ou bien si on limite exclusivement leur accès aux méthodes implémentant ces attributs.

11.2 Données de classe prêtes à l'emploi

Lorsqu'on se trouve devant un problème difficile, le plus simple est de laisser un autre le résoudre à votre place ! Si c'est le cas, `Class::Data::Inheritable` (disponible sur un serveur CPAN près de chez vous) offre une solution "clé en main" au problème des données de classe, solution qui utilise les fermetures. Aussi, avant de vous égarer dans ce document, je vous conseille de jeter un coup d'oeil à ce module.

11.3 Données de classe en tant que variables de paquetage

Le choix le plus naturel est d'utiliser les variables de paquetage pour stocker les attributs de classe, tout simplement parce qu'une classe Perl est un paquetage. Ceci simplifie, pour chaque classe, l'implémentation de ses propres attributs de classe. Supposons que l'on ait une classe nommée `Some_Class`, nécessitant une paire d'attributs différents que vous voulez rendre accessibles à la classe entière. La chose la plus simple à faire est d'utiliser pour ces attributs des variables de paquetage telles que `$Some_Class::CData1` et `$Some_Class::CData2`. Mais nous n'encouragerons pas l'utilisateur à utiliser directement ces données, c'est pourquoi nous lui fournirons des méthodes d'accès à ces variables.

Dans les méthodes d'accès ci-dessous, nous allons pour le moment ignorer le premier argument - la partie située à gauche de la flèche de l'appel de la méthode, qui est soit un nom de classe soit une référence d'objet.

```
package Some_Class;
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $Some_Class::CData1 = shift if @_;
    return $Some_Class::CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $Some_Class::CData2 = shift if @_;
    return $Some_Class::CData2;
}
```

Cette technique est très claire, et devrait être très simple à mettre en oeuvre même pour un novice de la programmation Perl. En qualifiant complètement les variables de paquetage, celles-ci apparaissent clairement à la lecture du code. Malheureusement, si l'on orthographie mal l'une d'entre elles, on introduit une erreur difficile à localiser. Par ailleurs, il est parfois déconcertant de trouver le nom de la classe codé "en dur" aussi souvent.

Ces deux problèmes trouvent facilement leur solution. Il suffit simplement d'ajouter le pragma `use strict`, puis de pré-déclarer les variables de paquetage. (l'opérateur `our` a vu le jour dans la version 5.6 de Perl, et représente pour les variables globales du paquetage ce que `my` représente pour les variables lexicales.)

```
package Some_Class;
use strict;
our($CData1, $CData2); # our() est nouveau depuis perl5.6
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData2 = shift if @_;
    return $CData2;
}
```

Tout comme pour les autres variables globales, certains programmeurs préfèrent mettre en majuscule la première lettre de leurs variables de paquetage. C'est relativement clair, mais si l'on ne qualifie pas complètement les variables de paquetage, on risque de perdre leur signification à la lecture du code. On peut facilement pallier à cela en choisissant de meilleurs noms.

11.3.1 Mettre tous ses oeufs dans le même panier

L'énumération des méthodes d'accès aux données de classe devient rapidement ennuyeuse, de manière similaire à l'énumération des méthodes d'accès aux attributs d'instance (voir *perltoot*). Cette répétition contredit la première vertu du programmeur, la paresse, qui se manifeste ici par le désir inné du programmeur de factoriser le code dupliqué partout où cela est possible.

Voici ce que nous allons faire. Tout d'abord, créer un simple hash qui contiendra tous les attributs de classe.

```
package Some_Class;
use strict;
our %ClassData = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);
```

En utilisant les fermetures (voir *perlref*) ainsi que l'accès direct à la table de symboles du paquetage (voir *perlmod*), nous allons cloner une méthode d'accès pour chaque clé du hash %ClassData. Chacune de ces méthodes évaluera ou modifiera la valeur d'un attribut de classe spécifique nommé.

```
for my $datum (keys %ClassData) {
    no strict "refs";      # pour enregistrer les nouvelles méthodes dans le paquetage
    *$datum = sub {
        shift; # XXX: ignorer l'appel sur une classe/objet
        $ClassData{$datum} = shift if @_;
        return $ClassData{$datum};
    }
}
```

En réalité nous pourrions utiliser une solution utilisant une méthode &AUTOLOAD, mais cette approche est loin d'être satisfaisante. La fonction devrait distinguer les attributs d'objets des attributs de classe ; elle pourrait interférer avec l'héritage ; et elle devrait tenir compte de DESTROY. Une telle complexité n'est pas nécessaire dans la plupart des cas, et certainement pas dans celui-ci.

Pourquoi l'utilisation de `no strict refs` dans la boucle ? Nous sommes en train de manipuler la table des symboles pour introduire de nouveaux noms de fonctions en utilisant des références symboliques (nommage indirect), ce qu'aurait interdit le pragma `strict`. Normalement, les références symboliques représentent au mieux une manière d'esquiver la question. Ce n'est pas uniquement dû au fait qu'elles puissent être utilisées accidentellement alors que vous n'y pensiez même pas. C'est aussi parce qu'il existe de bien meilleures approches pour les nombreux cas où les programmeurs Perl débutants tentent d'utiliser les références symboliques, par exemple les hash imbriqués ou les hash de tableaux. Mais il n'est pas interdit d'utiliser les références symboliques pour manipuler quoi que ce soit d'intéressant du point de vue de la table des symboles du paquetage, par exemple les noms de méthodes ou les variables de paquetage. En d'autres termes, utilisez des références de symbole lorsque vous désirez vous référer à la table des symboles.

Il y a plusieurs avantages à confiner tous les attributs de classe en un même lieu. On peut aisément les visualiser, les initialiser ou les modifier. Ceci les rend également plus facile d'accès depuis l'extérieur, par exemple depuis un débogueur ou un paquetage persistant. Reste un seul problème, nous n'avons pas automatiquement connaissance du nom de chaque objet d'une classe, même s'il en a un. Ceci est traité plus loin dans Le méta-objet éponyme (§11.3.3).

11.3.2 À propos de l'héritage

Supposons que nousinstancions une classe dérivée, et que nous accédons à ses données de classe au moyen de l'appel d'une méthode héritée. Allons-nous récupérer les attributs de la classe de base, ou bien ceux de la classe dérivée ? Que se passera-t-il dans les exemples précédents ? La classe dérivée hérite de toutes les méthodes de la classe de base, y compris celles qui accèdent aux attributs de classe. Mais à quel paquetage appartiennent les attributs de classe ?

Ainsi que le suggère leur nom, la réponse est que les attributs de classe sont stockés dans le paquetage où ces méthodes ont été compilées. Lorsqu'on invoque la méthode `&CData1` sur le nom de la classe dérivée ou sur celui d'un objet de cette classe, la version vue précédemment est toujours valable, et donc on accédera à `$Some_Class::CData1` - ou, dans la version clonant les méthodes, à `$Some_Class::ClassData{CData1}`.

Il faut garder à l'esprit que ces méthodes de classe s'exécutent dans le contexte de leur classe de base, et non dans celui de la classe dérivée. Parfois c'est exactement ce que nous voulons. Si `Feline` est une sous-classe de `Carnivore`, alors la

population mondiale des Carnivore s'accroît à chaque naissance d'un Feline. Mais qu'en est-il si l'on cherche combien il y a de Feline parmi les Carnivore ? L'approche actuelle ne répond pas à cette question.

Il faut décider, au cas par cas, si le fait que les attributs de classe soient relatifs au paquetage a un sens. Si c'est votre choix, alors il ne faut plus laisser de côté le premier argument de la fonction. Ce peut être aussi bien un nom de paquetage si la méthode a été invoquée directement sur un nom de paquetage, ou bien une référence d'objet si la méthode a été invoquée sur une référence d'objet. Auquel cas la fonction `ref()` renvoie la classe de cet objet.

```
package Some_Class;
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::CData1";
    no strict "refs";          # pour accéder symboliquement aux données du paquetage
    $$varname = shift if @_;
    return $$varname;
}
```

Il faut alors faire de même pour tous les autres attributs de classe (tels que `CData2`, etc.) pour lesquels vous souhaitez un accès aux variables du paquetage invoqué plutôt qu'un accès au paquetage compilé que nous avons précédemment.

Une fois de plus nous désactivons temporairement les références strictes, faute de quoi nous ne pourrions pas utiliser les noms symboliques pleinement qualifiés pour le paquetage global. Ceci est très raisonnable : puisque les variables de paquetage résident par définition dans un paquetage, rien n'interdit d'y accéder via la table de symboles de ce paquetage. C'est d'ailleurs la raison de sa présence (Bon, ça suffit).

Que penser si nous utilisons un simple hash à tout faire, ainsi que des méthodes clonées ? A quoi cela ressemblerait-il ? La seule différence résiderait dans les fermetures utilisées pour générer de nouvelles entrées de méthodes pour la table de symboles de la classe.

```
no strict "refs";
*$datum = sub {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::ClassData";
    $varname->{$datum} = shift if @_;
    return $varname->{$datum};
}
```

11.3.3 Le méta-objet éponyme

N.d.t : éponyme signifie "qui donne son nom". Le méta-objet (la classe) "donne son nom" au hash qui le représente.

On pourrait dire que le nom du hash `%ClassData` de l'exemple précédent n'est ni le plus imaginaire ni le plus intuitif. Peut-on trouver quelque chose de plus sensé, ou de plus utile, voire les deux ?

La réponse est, de fait, positive. Pour un "méta-objet de classe", nous allons utiliser une variable de paquetage de même nom que le paquetage lui-même. Dans la portée lexicale des déclarations d'un paquetage `Some_Class`, nous utiliserons le hash de même nom `%Some_Class` comme méta-objet de cette classe. (Utiliser un hash nommé de manière éponyme ressemble beaucoup aux classes nommant leurs constructeurs de manière éponyme, à la manière de Python ou de C++. Cela signifie que la classe `Some_Class` pourrait avoir un constructeur `Some_Class::Some_Class`, et sans doute exporter ce nom. Si vous cherchez un exemple, c'est ce que fait la classe `StrNum` de la recette 13.14 du *Perl Cookbook*.)

Cette approche prévisible est très appréciable, entre autres parce qu'elle fournit un identificateur connu pour l'aide au débogging, à la persistance transparente, ou au contrôle. C'est également le nom évident pour les classes monadiques et les attributs transparents que nous aborderons plus tard.

Voici un exemple d'une telle classe. Remarquez le nom du hash contenant le méta-objet, le même que celui du paquetage utilisé pour implémenter la classe.

```
package Some_Class;
use strict;
```

```

# créer un méta-objet de classe en utilisant ces noms si merveilleux
our %Some_Class = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

# cette méthode d'accès est relative au package appelant
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    no strict "refs";      # pour accéder au méta-objet éponyme
    $class->{CData1} = shift if @_;
    return $class->{CData1};
}

# but this accessor is not
sub CData2 {
    shift;                # XXX: ignorer l'appel sur une classe/objet
    no strict "refs";     # pour accéder au méta-objet éponyme
    __PACKAGE__ -> {CData2} = shift if @_;
    return __PACKAGE__ -> {CData2};
}

```

Dans la deuxième méthode d'accès, la notation `__PACKAGE__` a été utilisée pour deux raisons. La première, pour éviter de coder en dur le nom du paquetage, au cas où nous déciderions ultérieurement de changer ce nom. La deuxième, pour éclairer le lecteur sur le fait qu'il s'agit du paquetage actuellement compilé, et non du paquetage de l'objet ou de la classe appelante. Si la longue séquence de caractères non alphabétiques vous dérange, on peut toujours créer une variable contenant la chaîne `__PACKAGE__`.

```

sub CData2 {
    shift;                # XXX: ignorer l'appel sur une classe/objet
    no strict "refs";     # pour accéder au méta-objet éponyme
    my $class = __PACKAGE__;
    $class->{CData2} = shift if @_;
    return $class->{CData2};
}

```

Bien que nous utilisions au mieux les références symboliques, certains seront déconcertés de voir le test sur les références strictes aussi souvent désactivé. Étant donné une référence symbolique, on peut toujours en déduire une référence réelle (bien que l'inverse ne soit pas vrai). Aussi nous allons écrire un sous-programme qui réalisera cette conversion pour nous. Appelé comme une fonction sans arguments, il renvoie une référence sur le hash éponyme de la classe compilée. Appelé en tant que méthode de classe, il renvoie une référence sur le hash éponyme du code appelant. Enfin, appelé en tant que méthode d'objet, il renvoie une référence vers le hash éponyme de toute classe à laquelle appartient l'objet.

```

package Some_Class;
use strict;

our %Some_Class = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

# ce sous-programme a trois faces : fonction, méthode de classe, ou méthode d'objet
sub _classobj {
    my $obclass = shift || __PACKAGE__;
    my $class   = ref($obclass) || $obclass;
    no strict "refs";    # pour convertir les références symboliques en références réelles
    return \%$class;
}

```

```

for my $datum (keys %{ _classobj() } ) {
    # inhiber "strict refs" pour pouvoir
    # enregistrer une méthode dans la table des symboles
    no strict "refs";
    *$datum = sub {
        use strict "refs";
        my $self = shift->_classobj();
        $self->{$datum} = shift if @_;
        return $self->{$datum};
    }
}

```

11.3.4 Références indirectes aux données de classe

Une stratégie classique et raisonnable pour se souvenir des attributs de classe est de garder une référence de chaque variable de paquetage dans l'objet lui-même. C'est une stratégie que vous avez probablement déjà rencontrée, par exemple dans *perltoot* et *perlot*, mais voici quelques variations auxquelles vous n'auriez sans doute pas pensé auparavant.

```

package Some_Class;
our($CData1, $CData2);          # our() est nouveau depuis perl5.6

sub new {
    my $obclass = shift;
    return bless my $self = {
        ObData1 => "",
        ObData2 => "",
        CData1  => \$CData1,
        CData2  => \$CData2,
    } => (ref $obclass || $obclass);
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    my $dataref = ref $self
                                ? $self->{CData1}
                                : \$CData1;

    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    my $dataref = ref $self
                                ? $self->{CData2}
                                : \$CData2;

    $$dataref = shift if @_;
    return $$dataref;
}

```

Comme vous le voyez ci-dessus, une classe dérivée héritera de ces méthodes, et pourra par conséquent accéder aux variables de paquetage dans le paquetage de la classe de base. Ce n'est pas nécessairement le comportement souhaité dans tous les cas. Voici un exemple utilisant un méta-objet variable, prenant soin d'accéder aux données du bon paquetage.

```

package Some_Class;
use strict;

our %Some_Class = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

sub _classobj {
    my $self = shift;
    my $class = ref($self) || $self;
    no strict "refs";
    # prendre une référence (hard) sur le meta-object éponyme
    return \%$class;
}

sub new {
    my $obclass = shift;
    my $classobj = $obclass->_classobj();
    bless my $self = {
        ObData1 => "",
        ObData2 => "",
        CData1  => \%$classobj->{CData1},
        CData2  => \%$classobj->{CData2},
    } => (ref $obclass || $obclass);
    return $self;
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData1};
    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData2};
    $$dataref = shift if @_;
    return $$dataref;
}

```

Le fait d'utiliser un méta-objet éponyme semble rendre le code plus propre, en plus de rendre l'utilisation de `strict refs` plus claire. A l'inverse de la version précédente, cette dernière nous montre quelque chose d'intéressant par rapport à l'héritage : elle accède au méta-objet de classe de la classe appelante plutôt qu'à celui appartenant à la classe dans laquelle la méthode a été initialement compilée.

Il est facile d'accéder aux données du méta-objet de classe, et de visualiser l'état de la classe dans son ensemble, en utilisant un mécanisme externe similaire à ceux utilisés dans les debugger ou lorsqu'on implémente une classe persistante. Cela fonctionne parce que le méta-objet de classe est une variable de paquetage, possède un nom connu de tous, et regroupe toutes ses données. (La persistance transparente n'est pas toujours réalisable, mais c'est certainement une idée attirante.)

Il n'y a pas de contrôle vérifiant si les méthodes d'accès aux objets ont été invoquées sur un nom de classe. Si le pragma `strict ref` est activé, ce problème n'existe pas. Sinon, vous obtiendrez le méta-objet éponyme. Ce que vous faites avec - ou à partir de - lui, c'est votre affaire. Les deux prochaines sections montrent de nouvelles utilisations de cette puissante caractéristique.

11.3.5 Les classes univalentes

Un certain nombre de modules standards livrés avec Perl fournissent des interfaces de classe dépourvus de méthodes d'attribut. Le module le plus communément utilisé parmi les pragma, le module `Exporter`, est une classe sans constructeur ni attributs. Son travail consiste simplement à fournir une interface standard pour les modules désireux d'exporter une partie de leur espace de nom dans celui de l'appelant. Les modules utilisent la méthode `&import` d'`Exporter` simplement en ajoutant la mention "Exporter" dans `@ISA`, le tableau de leur paquetage décrivant la liste des ancêtres. Cependant la classe "Exporter" ne fournit pas de constructeur, on ne peut donc avoir plusieurs instances de cette classe. En fait, on ne peut en avoir aucune - cela n'aurait pas de sens. Nous n'avons accès qu'à ses méthodes. L'interface ne contient pas d'information d'état, aussi les données d'état sont totalement superflues.

On voit de temps en temps une classe qui n'accepte qu'une instance unique. Ce type de classe s'appelle une *classe univalente*, ou, moins formellement, un *singleton* ou une *classe des hautes-terres*.

Où ranger l'état, les attributs d'une classe monadique ? Comment être certain qu'il n'existera jamais plus d'une instance ? Là où l'on pourrait simplement utiliser un groupe de variables de paquetage, il est tout de même plus propre d'utiliser le hash du même nom. Voici un exemple complet de classe monadique :

```
package Cosmos;
%Cosmos = ();

# méthode d'accès pour l'attribut "name"
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}

# méthode d'accès en lecture seulement pour l'attribut "birthday"
sub birthday {
    my $self = shift;
    die "impossible de modifier la date de naissance" if @_; # XXX: croak() serait mieux
    return $self->{birthday};
}

# méthode d'accès pour l'attribut "stars"
sub stars {
    my $self = shift;
    $self->{stars} = shift if @_;
    return $self->{stars};
}

# rogneugneu ! - une de nos étoiles vient de se faire la belle !
sub supernova {
    my $self = shift;
    my $count = $self->stars();
    $self->stars($count - 1) if $count > 0;
}
```

```

# méthode constructeur/initialiseur - redémarrage
sub bigbang {
    my $self = shift;
    %$self = (
        name      => "the world according to tchrist",
        birthday   => time(),
        stars      => 0,
    );
    return $self;      # oui, il s'agit probablement d'une classe. SURPRISE !
}

# Après que la classe ait été compilée, mais avant le retour d'un quelconque
# use ou require, démarrage de l'univers par un bang.
__PACKAGE__ -> bigbang();

```

Un instant : ceci n'a pas l'air bien extraordinaire. Ces accesseurs d'attributs de classe univalente ne semblent pas différents de ceux d'une classe habituelle. Le point important est que rien n'indique que `$self` doit être une référence à un objet consacré. Simplement ce doit être quelque chose sur lequel on peut invoquer des méthodes. Ici le nom du paquetage lui-même, `Cosmos`, fonctionne comme un objet. Regardez la méthode `&supernova`. Est-ce une méthode de classe ou une méthode d'objet ? L'analyse statique ne peut pas donner la réponse. Perl ne s'intéresse pas à cela, et c'est ce que vous devriez faire également. Dans les trois méthodes d'attributs, `__$self` accède vraiment aux variables de paquetage de `%Cosmos`.

Si, comme Stephen Hawking, vous érigez en principe l'existence d'univers multiples, séquentiels et sans aucun rapport entre eux, alors vous pouvez invoquer la méthode `&bigbang` à n'importe quel moment afin de tout recommencer. Vous pourriez penser à la méthode `&bigbang` davantage comme à un initialiseur qu'à un constructeur, dans la mesure où la fonction n'alloue pas de mémoire supplémentaire ; elle initialise simplement tout ce qui existe déjà. Seulement, comme n'importe quel autre constructeur, elle retourne une valeur scalaire que l'on utilisera ultérieurement dans les invocations de méthodes.

Supposez que quelque temps après vous décidez qu'un seul univers n'est pas suffisant. Vous pourriez réécrire entièrement une nouvelle classe, or vous possédez déjà une classe qui fait tout ce que voulez - sauf qu'elle est univalente, et que vous voulez plusieurs cosmos.

La réutilisation du code via l'héritage de classe n'est pas autre chose. Regardez la taille du nouveau code :

```

package Multiverse;
use Cosmos;
@ISA = qw(Cosmos);

sub new {
    my $protoverse = shift;
    my $class      = ref($protoverse) || $protoverse;
    my $self       = {};
    return bless($self, $class)->bigbang();
}
1;

```

Ayant soigneusement défini la classe `Cosmos` lors de sa création, nous pouvons le moment venu la réutiliser sans toucher à une seule ligne de code afin d'écrire notre classe `Multiverse`. Le code qui fonctionnait lorsqu'on l'invoquait en tant que méthode de classe continue à fonctionner parfaitement lorsqu'il est invoqué sur chacune des instances de la classe dérivée.

On peut s'étonner que dans la classe `Cosmos` ci-dessus la valeur retournée par le "constructeur" `&bigbang` ne soit pas une référence à un objet consacré. C'est tout simplement le propre nom de la classe. Un nom de classe est de fait un objet parfaitement acceptable. Il a un état, une signification, une identité, les trois attributs cruciaux d'un système objet. Il ouvre la porte à l'héritage, au polymorphisme et à l'encapsulation. Que demander de plus à un objet ?

Pour comprendre l'orientation objet en Perl, il est important de reconnaître l'unification en simple méthodes de ce que d'autres langages pourraient appeler "méthodes de classe" et "méthodes d'objet". Les "méthodes de classe" et les "méthodes d'objet" sont différenciées uniquement dans l'esprit du programmeur Perl, mais non dans le langage Perl lui-même.

Ceci étant, un constructeur n'a pas de caractéristiques spéciales, et c'est une des raisons pour lesquelles Perl n'a pas de mot-clé pour le désigner. Un "constructeur" est simplement un terme informel utilisé vaguement pour décrire une méthode retournant une valeur scalaire sur laquelle on pourra ensuite appliquer des appels de méthode. Il suffit de savoir que cette

valeur scalaire peut aussi bien être un nom de classe qu'une référence d'objet. Il n'y a aucune raison pour qu'elle soit une référence sur un objet flambant neuf.

Il peut exister autant - ou aussi peu - de constructeurs que l'on veut, et leur nom importe peu. Nommer aveuglément et docilement `new()` chaque nouveau constructeur que l'on écrit signifie que l'on parle le Perl avec un fort accent C++, ce qui dessert les deux langages. Il n'y a aucune raison pour que chaque classe n'ait qu'un constructeur, ou bien pour qu'un constructeur s'appelle `new()`, ou encore qu'un constructeur soit utilisé en tant que méthode de classe et non en tant que méthode d'objet.

La section suivante montre l'utilité de prendre de la distance par rapport à la distinction formelle entre les appels aux méthodes de classe et ceux des méthodes d'objet, aussi bien dans les constructeurs que dans les méthodes d'accès.

11.3.6 Les attributs transparents

Le rôle du hash éponyme de paquetage ne se résume pas uniquement à recueillir une classe ainsi que des données d'état globales. Il peut aussi représenter une sorte de modèle contenant les valeurs initiales par défaut des attributs d'objet. Ces valeurs initiales peuvent alors être utilisées dans les constructeurs pour initialiser un objet particulier. Il peut aussi servir à implémenter les attributs transparents. Un attribut transparent possède une valeur par défaut valable pour toute la classe. Chaque objet peut lui donner une valeur, et dans ce cas `$object->attribute()` retournera cette valeur. Mais si la valeur n'a pas été initialisée, `$object->attribute()` retournera la valeur par défaut pour la classe.

Ce comportement nous permet d'avoir une approche du type "copie-si-écriture" pour ces attributs transparents. Si l'on se contente de lire leur valeur, tout reste transparent. Mais si on modifie leur valeur, cette nouvelle valeur n'est valable que pour l'objet courant. D'autre part, si l'on utilise la classe en tant qu'objet et qu'on y modifie directement la valeur d'un attribut, la valeur du méta-objet sera modifiée, et par suite des lectures ultérieures sur les objets qui n'auront pas initialisé ces attributs retourneront les nouvelles valeurs du méta-objet. Tandis que les objets qui auront initialisé la valeur de l'attribut ne verront aucun changement.

Voyons un exemple concret utilisant cette fonctionnalité avant de monter la manière d'implémenter ces attributs. Soit une classe de nom `Some_Class` possédant un attribut transparent nommé `color`. Initialisons d'abord la couleur dans le méta-objet, puis créons trois objets au moyen d'un constructeur nommé par exemple `&spawn`.

```
use Vermin;
Vermin->color("vermilion");

$ob1 = Vermin->spawn(); # c'est ainsi qu'arriva le Jedi
$ob2 = Vermin->spawn();
$ob3 = Vermin->spawn();

print $obj3->color(); # affiche "vermilion"
```

Chacun de ces objets est maintenant de couleur "vermilion", parce que "vermilion" est la valeur de cet attribut dans le méta-objet, mais ces objets ne possèdent pas de couleur individuelle initialisée.

Si l'on modifie la valeur d'un objet, cela n'a aucun effet sur les autres objets créés auparavant.

```
$ob3->color("chartreuse");
print $ob3->color(); # affiche "chartreuse"
print $ob1->color(); # affiche "vermilion", de manière transparente
```

Si maintenant on utilise `$ob3` pour créer un autre objet, le nouvel objet héritera de sa couleur, qui sera maintenant "chartreuse". La raison de ceci est que le constructeur utilise l'objet invoqué comme modèle pour les initialisations d'attributs. Lorsque l'objet invoqué est le nom de classe, l'objet utilisé comme modèle est le méta-objet éponyme. Lorsque l'objet invoqué est une référence sur un objet instancié, le constructeur `&spawn` se sert de lui comme modèle.

```
$ob4 = $ob3->spawn(); # $ob3 est maintenant un modèle, et non %Vermin
print $ob4->color(); # affiche "chartreuse"
```

Toute valeur réelle initialisée dans l'objet modèle sera copiée dans le nouvel objet. Mais les attributs non définis dans l'objet modèle, transparents par définition, le resteront dans le nouvel objet.

Modifions maintenant l'attribut de couleur de l'ensemble de la classe :

```

Vermin->color("azure");
print $ob1->color();   # affiche "azure"
print $ob2->color();   # affiche "azure"
print $ob3->color();   # affiche "chartreuse"
print $ob4->color();   # affiche "chartreuse"

```

Ce changement de couleur n'a d'effet que pour les deux premiers objets, ceux qui étaient transparents lors d'un accès aux valeurs du méta-objet. Les couleurs des deux suivants étaient déjà initialisées, et par conséquent ne changent pas.

Reste une question importante. Les modifications du méta-objet sont propagées dans les attributs transparents de la classe entière, mais qu'en est-il des modifications effectuées sur les objets particuliers ? Si l'on modifie la couleur de \$ob3, celle de \$ob4 est-elle modifiée ? Ou, inversement, si l'on modifie celle de \$ob4, celle de \$ob3 s'en trouve-t-elle modifiée ?

```

$ob3->color("amethyst");
print $ob3->color();   # affiche "amethyst"
print $ob4->color();   # hmm: "chartreuse" ou "amethyst"?

```

Il ne faut pas faire cela, bien que certains affirment que nous le devrions dans certaines situations assez rares. La réponse à la question posée dans le commentaire ci-dessus, mis à part le bon goût, doit être "chartreuse" et non "amethyst". Ainsi nous devons traiter ces attributs un peu à la manière dont les attributs de processus (tels que les variables d'environnement, les identificateurs d'utilisateur ou de groupe, ou encore le répertoire courant) le sont lors d'un `fork()`. Vous seul pouvez les modifier, mais gardez à l'esprit que ces modifications se retrouveront dans les fils à naître. Les modifications d'un objet ne seront jamais répercutées vers leur parent ni vers d'éventuels objets fils existants. Toutefois ces modifications s'appliqueront à de tels objets créés ultérieurement.

Que faire pour un objet possédant des valeurs réelles pour ses attributs, et dont on veut rendre les valeurs d'attribut d'objet à nouveau transparentes ? Nous allons définir la classe de manière que cet attribut redevienne transparent lorsqu'on invoque une méthode d'accès avec un argument `undef`.

```

$ob4->color(undef);           # retour vers "azure"

```

Voici l'implémentation complète de `Vermin` telle que décrite ci-dessus.

```

package Vermin;

# voici le méta-objet de classe, nommé eponymement.
# il contient tous les attributs de classe, ainsi que tous les attributs d'instances
# c'est ainsi que la suite peut être utilisée pour les deux initialisations
# et la transparence.

our %Vermin = (
    PopCount => 0,          # majuscules pour un attribut de classe
    color    => "beige",    # minuscules pour un attribut d'instance
);

# méthode "constructeur"
# invoquée en tant que méthode de classe ou méthode d'objet
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $class->{PopCount}++;
    # initialiser les champs à partir de l'objet appelant, ou les omettre
    # si l'objet appelant est la classe, afin de fournir la transparence
    %$self = %$obclass if ref $obclass;
    return $self;
}

# méthode d'accès transparente pour l'attribut "color"
# invoquée en tant que méthode de classe ou méthode d'objet
sub color {
    my $self = shift;
    my $class = ref($self) || $self;

```



```

    # prend en charge l'invocation de classe
    unless (ref $self) {
        $class->{color} = shift if @_;
        return $class->{color}
    }

    # prend en charge l'invocation depuis un objet
    $self->{color} = shift if @_;
    if (defined $self->{color}) { # not exists!
        return $self->{color};
    } else {
        return $class->{color};
    }
}

# methode d'accès pour l'attribut de classe "PopCount"
# invoquée en tant que méthode de classe ou méthode d'objet
# mais utilise l'objet uniquement pour repérer le méta-objet
sub population {
    my $obclass = shift;
    my $class = ref($obclass) || $obclass;
    return $class->{PopCount};
}

# destructeur d'instance
# invoqué uniquement en tant que méthode d'objet
sub DESTROY {
    my $self = shift;
    my $class = ref $self;
    $class->{PopCount}--;
}

```

Voici une paire de méthodes d'aide assez pratiques. Ce ne sont nullement des méthodes d'accès. Elles sont utilisées pour détecter l'accessibilité des attributs. La méthode `&is_translucent` détermine si un attribut d'un objet particulier provient du méta-objet. La méthode `&has_attribute` détecte si une classe implémente une propriété particulière. Elle peut également servir à déterminer si une propriété est indéfinie ou si elle n'existe pas.

```

# détermine si un attribut d'objet est transparent
# (typiquement ?) invoquée comme une méthode d'objet
sub is_translucent {
    my($self, $attr) = @_;
    return !defined $self->{$attr};
}

# teste la présence d'un attribut dans la classe
# invoquée en tant que méthode de classe ou méthode d'objet
sub has_attribute {
    my($self, $attr) = @_;
    my $class = ref $self if $self;
    return exists $class->{$attr};
}

```

Si l'on préfère installer les méthodes d'accès de manière plus générique, on peut se servir de la convention majuscules vs minuscules pour enregistrer dans le paquetage les méthodes appropriées clonées depuis les fermetures génériques.

```

for my $datum (keys %{ +__PACKAGE__ }) {
    *$datum = ($datum =~ /^[A-Z]/)
        ? sub { # installe la méthode d'accès de classe
            my $obclass = shift;
            my $class = ref($obclass) || $obclass;

```

```

        return $class->{$datum};
    }
    : sub { # installe la méthode d'accès transparente
        my $self = shift;
        my $class = ref($self) || $self;
        unless (ref $self) {
            $class->{$datum} = shift if @_;
            return $class->{$datum}
        }
        $self->{$datum} = shift if @_;
        return defined $self->{$datum}
            ? $self -> {$datum}
            : $class -> {$datum}
    }
}

```

En guise d'exercice, nous proposons au lecteur de traduire en C++, Java et Python cette approche basée sur les fermetures. N'oubliez pas de me prévenir par mail dès que vous aurez réalisé cela.

11.4 Données de classe en tant que variables lexicales

11.4.1 Domaine privé et responsabilité

Dans les exemples qui précèdent, à l'inverse des conventions utilisées par certains programmeurs Perl, nous n'avons pas préfixé au moyen d'un caractère souligné les variables de paquetage servant d'attribut de classe, pas davantage que les noms de clés utilisées pour les attributs d'instance. Nous n'avons pas besoin de ces marqueurs pour suggérer une propriété de fait sur les variables d'attributs ou les clés de hash, parce qu'elles sont toujours notoirement privées ! Le monde extérieur n'a pas à jouer avec quoi que ce soit dont le moyen d'accès a été publié par une classe au travers d'une interface documentée ; en d'autres termes, au moyen des invocations de méthode. Et pas davantage au moyen de n'importe quelle méthode. Les méthodes dont le nom commence par un souligné sont traditionnellement considérées comme étant à la limite extérieure de la classe. Si des étrangers ignorent l'interface des méthodes documentées et jouent avec les variables internes de votre classe de sorte qu'à la fin quelque chose casse, ce n'est pas votre faute - c'est la leur.

Perl préfère la responsabilité individuelle au contrôle imposé. Perl vous respecte suffisamment pour vous laisser choisir votre niveau préféré de peine ou de plaisir. Perl suppose que vous êtes créatif, intelligent, capable de prendre vos propres décisions - et attend de vous que vous preniez l'entière responsabilité de vos actions. Dans un monde parfait, ces remontrances devraient être suffisantes, et tout le monde serait intelligent, responsable, heureux, et créatif. Et soigneux. Il ne faut surtout pas oublier d'être soigneux, mais il est quelque peu difficile d'espérer cela. Même Einstein peut prendre accidentellement un mauvais virage et se retrouver finalement perdu dans un quartier inconnu de la ville.

Certains deviennent paranoïaques à la vue de variables de paquetage exposées à la vue de tous ceux qui pourraient les atteindre et les altérer. Certains vivent dans la crainte constante que n'importe qui puisse réaliser une méchanceté n'importe où. La solution à ce problème est bien entendu de tuer le méchant. Malheureusement ce n'est pas aussi simple que cela. Ces personnes précautionneuses sont également effrayées par le fait qu'elles-mêmes ou d'autres puissent réaliser cela non par méchanceté mais par manque de soin, aussi bien par accident que par désespoir. Mais si l'on punit tous les maladroits, très rapidement plus personne ne pourra réaliser quoi que ce soit.

Si la paranoïa ou les précautions sont inutiles, ce manque de facilité peut représenter un problème pour certains. Celui-ci peut être allégé si l'on prend l'option de stocker les attributs de classe en tant que variables lexicales plutôt qu'en tant que variables de paquetage. L'opérateur `my()` est la source de toute propriété en Perl, et c'est en effet une forme de propriété très puissante.

Il est bien connu, et cela a souvent été écrit, que Perl ne permet pas de cacher les variables, ce qui n'offre aucune intimité ni aucun isolement au concepteur de classe, et au lieu de cela simplement un fragile ramassis de conventions sociales. Il est facile de prouver que cette perception est fautive. Dans la section suivante, nous montrerons comment implémenter des formes de propriété aussi puissantes que celles fournies dans la majorité des autres langages orientés-objet.

11.4.2 Variables lexicales dans la portée du fichier

Une variable lexicale n'est visible que jusqu'à la fin de sa portée statique. Cela signifie que le seul code capable d'y accéder est celui qui réside textuellement après l'opérateur `my()` et avant la fin du bloc courant s'il existe, ou jusqu'à la fin du fichier courant s'il n'existe pas.

En repartant de l'exemple le plus simple présenté au début de ce document, remplaçons les variables our() par leur version my().

```
package Some_Class;
my($CData1, $CData2); # la portée est le fichier, et dans un quelconque paquetage
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData2 = shift if @_;
    return $CData2;
}
```

Assez de cette ancienne variable de paquetage `$Some_Class::CData1` et de ses frères ! Elles ont maintenant disparu, remplacées par des lexicales. Personne, en dehors de la portée de la classe, ne peut les atteindre ni modifier l'état de la classe autrement qu'en utilisant l'interface documentée. Même les sous-classes ou les superclasses de celle-ci n'ont pas d'accès direct à `$CData1`. Elles doivent invoquer les méthodes d'accès à `$CData1` sur `Some_Class` ou sur une instance de cette classe, exactement comme tout le monde.

Pour être tout à fait honnête, la dernière affirmation suppose que l'on n'a pas placé les paquetages de diverses classes dans le même fichier, ni implémenté la classe dans plusieurs fichiers différents. L'accessibilité de ces variables est basée uniquement sur la portée statique du fichier. Elle n'a rien à voir avec le paquetage. Cela signifie concrètement que le code appartenant à la même classe mais se trouvant dans un fichier différent n'a pas d'accès à ces variables, alors que le code situé dans le même fichier mais appartenant à un autre paquetage (une autre classe) le peut. C'est pourquoi on suggère habituellement de mettre chaque module, classe, ou paquetage dans un seul fichier. Vous n'êtes pas obligé de suivre cette suggestion si vous savez vraiment ce que vous faites, cependant vous pourriez parfois vous emmêler les pincesaux, particulièrement au début.

Vous êtes parfaitement libres de grouper vos attributs de classe dans une structure composite de portée lexicale.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $ClassData{CData1} = shift if @_;
    return $ClassData{CData1};
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $ClassData{CData2} = shift if @_;
    return $ClassData{CData2};
}
```

Afin de simplifier les choses lorsqu'on ajoute d'autres attributs de classe, on peut créer des méthodes d'accès en enregistrant des fermetures dans la table de symboles du paquetage.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
for my $datum (keys %ClassData) {
    no strict "refs";
    *$datum = sub {
        shift; # XXX: ignorer l'appel sur une classe/objet
        $ClassData{$datum} = shift if @_;
    };
}
```

```

        return $ClassData{$datum};
    };
}

```

C'est certainement une bonne chose que d'obliger sa propre classe à utiliser les méthodes d'accès comme tout un chacun. Une meilleure est d'exiger et de s'attendre à ce que tous, y compris les sous-classes et les super-classes, les amis ou les ennemis, fassent de même. C'est un point critique du modèle. Il ne faut plus penser en termes de données "publiques" ou "protégées", notions séduisantes mais finalement destructrices. Les deux se retourneront contre vous. La raison en est que dès que vous ferez un pas hors de la position stable où tout est complètement privé, excepté dans la perspective des méthodes d'accès, vous aurez violé l'enveloppe. Et, ayant ouvert la boîte de Pandore de cette enveloppe d'encapsulation, vous en paierez sans nul doute le prix lorsque de futures modifications de l'implémentation rendront inopérant un code qui n'a rien à voir avec cela. Si l'on pense que ce n'était pas votre objectif lorsque vous avez décidé de passer à l'orienté-objet, quitte à être bridé et à subir les flèches d'une abstraction obséquieuse, une telle rupture semble extrêmement malheureuse.

11.4.3 Retour sur l'héritage

Supposons que l'on dérive `Another_Class` à partir de la classe de base `Some_Class`. Qu'obtient-on en invoquant une des méthodes `&CData` sur la classe dérivée ou sur un objet de cette classe ? La classe dérivée aura-t-elle son propre état, ou bien héritera-t-elle de la version de la classe de base des attributs de classe ?

La réponse se trouve dans l'exposé ci-dessus : la classe dérivée n'aura **pas** ses propres données d'état. Ainsi que précédemment, on peut considérer cela comme une bonne ou une mauvaise chose en fonction de la sémantique de la classe en question.

Le moyen le plus propre, le plus sain et le plus simple que possède une classe dérivée pour se référer à son état dans sa portée lexicale est de surcharger la méthode de la classe de base qui accède aux attributs de classe. Puisque la méthode réellement appelée est celle d'un objet de la classe dérivée si ce dernier existe, on obtient automatiquement l'état de la classe dérivée de cette manière. Il faut refouler de manière énergique toute envie de fournir une méthode non publiée destinée à obtenir une référence sur le hash `%ClassData`.

Tout comme pour d'autres surcharges de méthode, l'implémentation dans la classe dérivée conserve la possibilité d'invoquer la version de la classe de base en plus de la sienne. En voici un exemple :

```

package Another_Class;
@ISA = qw(Some_Class);

my %ClassData = (
    CData1 => "",
);

sub CData1 {
    my($self, $newvalue) = @_;
    if (@_ > 1) {
        # d'abord, enregistrer localement
        $ClassData{CData1} = $newvalue;

        # ensuite, passer le relais à la première
        # version surchargée, si elle existe
        if ($self->can("SUPER::CData1")) {
            $self->SUPER::CData1($newvalue);
        }
    }
    return $ClassData{CData1};
}

```

Ces remarques sur l'héritage multiple conservent leur intérêt au-delà de la première surcharge.

```

for my $parent (@ISA) {
    my $methname = $parent . "CData1";
    if ($self->can($methname)) {
        $self->$methname($newvalue);
    }
}

```

Pour améliorer les performances de manière significative, il est possible d'utiliser directement la méthode `&UNIVERSAL::can` qui renvoie une référence directe à la fonction :

```
for my $parent (@ISA) {
    if (my $coderef = $self->can($parent . "::CData1")) {
        $self->$coderef($newvalue);
    }
}
```

Si vous surdéfinissez `UNIVERSAL::can` dans votre propre classe, assurez-vous de retourner correctement cette référence.

11.4.4 Fermer la porte et jeter la clé

Avec l'implémentation actuelle, tout code se trouvant dans la portée lexicale du fichier contenant `%ClassData` a la possibilité de modifier directement ce dernier. Est-ce correct ? Peut-on accepter d'autoriser, ou si vous préférez, peut-on souhaiter autoriser d'autres parties de l'implémentation de cette classe à accéder directement aux attributs de classe ?

Cela dépend de ce que vous entendez par "être soigneux". Pensez à la classe `Cosmos`. Si la méthode `&supernova` avait pu directement modifier `$Cosmos::Stars` ou `$Cosmos::Cosmos{stars}`, alors nous n'aurions pas été capables de réutiliser la classe pour créer `Multiiverse`. Ceci montre définitivement que ce n'est certainement pas une bonne idée que d'autoriser l'accès aux attributs de classe par la classe elle-même par un autre moyen que les méthodes d'accès.

On ne peut pas généralement pas imposer à la classe un accès restreint à ses attributs de classe, même dans les langages fortement orientés-objet. Mais c'est possible en Perl.

En voici une manière :

```
package Some_Class;

{ # ouvrir une portée lexicale pour cacher $CData1
    my $CData1;
    sub CData1 {
        shift; # XXX: inutilisé
        $CData1 = shift if @_;
        return $CData1;
    }
}

{ # ouvrir une portée lexicale pour cacher $CData2
    my $CData2;
    sub CData2 {
        shift; # XXX: inutilisé
        $CData2 = shift if @_;
        return $CData2;
    }
}
```

Personne - absolument personne - n'est autorisé à lire ou modifier les attributs de classe autrement qu'au moyen de la méthode d'accès, puisque seule cette méthode a accès à la variable lexicale qu'elle gère. Cette utilisation d'une méthode d'accès aux attributs de classe est une forme de propriété largement plus puissante que celles fournies par la plupart des langage OO.

La duplication du code utilisée pour les méthodes d'accès aux données irrite notre paresse, nous allons donc utiliser des fermetures pour créer des méthodes similaires.

```
package Some_Class;

{ # ouvrir une portée lexicale pour les méta-objets ultra-privés des attributs de classe
    my %ClassData = (
        CData1 => "",
        CData2 => "",
    );
}
```

```

    for my $datum (keys %ClassData ) {
        no strict "refs";
        *$datum = sub {
            use strict "refs";
            my ($self, $newvalue) = @_;
            $ClassData{$datum} = $newvalue if @_ > 1;
            return $ClassData{$datum};
        }
    }
}

```

La fermeture ci-dessus peut être modifiée afin d'hériter des méthodes `&UNIVERSAL::can` et `SUPER` ainsi qu'on l'a vu précédemment.

11.4.5 Retour sur la transparence

La classe `Vermin` montre la transparence au moyen d'une variable de paquetage, nommée éponymement `%Vermin`, comme son méta-objet. Il est impossible d'utiliser cette stratégie si l'on préfère ne pas utiliser de variables de paquetage hormis celles nécessaires à l'héritage ou éventuellement au module `Exporter`. Ce n'est pas très satisfaisant, parce que les attributs transparents représentent une technique attirante, aussi vaut-il mieux envisager une implémentation qui utilise uniquement les variables lexicales.

Une deuxième raison pourrait conduire à éviter les hash de paquetage éponymes. A trop utiliser des noms de classe contenant des caractères "deux-points" doublés, on finit par obtenir quelque chose que l'on n'a pas voulu.

```

package Vermin;
$class = "Vermin";
$class->{PopCount}++;
# accesés $Vermin::Vermin{PopCount}

package Vermin::Noxious;
$class = "Vermin::Noxious";
$class->{PopCount}++;
# accesés $Vermin::Noxious{PopCount}

```

Dans le premier cas, le nom de classe n'a pas de "double-deux-points", on obtient le hash du paquetage courant. Mais dans le deuxième cas, au lieu de cela, on obtient le hash `%Noxious` du paquetage `Vermin`. (La vermine nocive achève tout juste d'envahir un autre paquetage et de semer ses données autour d'elle :-). Perl ne connaît pas la notion de paquetage relatif dans ses conventions de nommage, aussi les double-deux-points impliquent une inspection pleinement qualifiée plutôt que confinée au paquetage courant.

Dans la pratique, il est peu probable que la classe `Vermin` possède une variable de paquetage nommée `%Noxious` que vous auriez vous-même créé. Si vous êtes très méfiant, vous pourriez toujours parier sur votre territoire dont vous connaissez les règles, par exemple en utilisant à la place les noms de classe `Eponymous::Vermin::Noxious` ou `Hieronymus::Vermin::Boschious` ou encore `Leave_Me_Alone::Vermin::Noxious`. Il est certain que l'existence d'une autre classe nommée `Eponymous::Vermin` possédant son propre hash `%Noxious` est théoriquement possible, et cela sera toujours vrai. Personne n'arbitre les noms de paquetage. On trouvera toujours un cas où une variable globale telle que `@Cwd::ISA` explosera si plus d'une classe se sert du même paquetage `Cwd`.

S'il vous reste encore un relent de paranoïa, il existe une autre solution. Rien n'oblige à utiliser une variable de paquetage pour un méta-objet de classe, ni pour une classe univalentes, ni pour les attributs transparents. Il suffit simplement de coder les méthodes afin qu'elles aient un accès lexical.

Voici une autre implémentation de la classe `Vermin` avec la sémantique utilisée précédemment, mais n'utilisant pas de variables de paquetage.

```

package Vermin;

# Voici le méta-objet de classe nommé éponymement.
# Il contient toutes les données de classe, ainsi que toutes les données d'instance
# on peut donc utiliser ce qui suit pour les deux initialisations
# ainsi que la transparence. C'est un modèle.
my %ClassData = (
    PopCount => 0,          # majuscules pour les attributs de classe
    color    => "beige",   # minuscules pour les attributs d'instance
);

```

```

# méthode "constructeur"
# invoquée en tant que méthode de classe ou méthode d'objet
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $ClassData{PopCount}++;
    # initialiser les champs à partir de l'objet appelant, ou les
    # omettre si l'objet appelant est la classe, afin de fournir la transparence
    %$self = %$obclass if ref $obclass;
    return $self;
}

# méthode d'accès transparente pour l'attribut "color"
# invoquée en tant que méthode de classe ou méthode d'objet
sub color {
    my $self = shift;

    # prend en charge l'invocation sur une classe
    unless (ref $self) {
        $ClassData{color} = shift if @_;
        return $ClassData{color}
    }

    # prend en charge l'invocation sur un objet
    $self->{color} = shift if @_;
    if (defined $self->{color}) { # cela n'existe pas !
        return $self->{color};
    } else {
        return $ClassData{color};
    }
}

# méthode d'accès d'attribut de classe pour l'attribut "PopCount"
# invoquée en tant que méthode de classe ou méthode d'objet
sub population {
    return $ClassData{PopCount};
}

# destructeur d'instance ; invoquée uniquement en tant que méthode d'objet
sub DESTROY {
    $ClassData{PopCount}--;
}

# détermine la transparence d'un attribut d'objet
# (typiquement ?) invoquée uniquement en tant que méthode d'objet
sub is_translucent {
    my($self, $attr) = @_;
    $self = \%ClassData if !ref $self;
    return !defined $self->{$attr};
}

# teste la présence d'un attribut dans la classe
# invoquée en tant que méthode de classe ou méthode d'objet
sub has_attribute {
    my($self, $attr) = @_;
    return exists $ClassData{$attr};
}

```

11.5 NOTES

L'héritage est un mécanisme très puissant mais très subtil, que l'on utilisera mieux après mûre réflexion. En revanche, l'agrégation représente souvent une meilleure approche.

Il n'est pas possible d'utiliser des variables dans la portée lexicale du fichier avec les modules `SelfLoader` ou `AutoLoader`, parce qu'ils modifient la portée lexicale dans laquelle s'expriment les méthodes du module lors de la compilation.

Il faudrait sans aucun doute prendre en compte les modifications de paquetage peu claires avant de créer les noms des attributs d'objet. Par exemple, `$self->{ObData1}` devrait plutôt s'appeler `$self->{ __PACKAGE__ . "_ObData1" }`, mais cela aurait rendu les exemples plus confus.

11.6 VOIR AUSSI

perltoot, *perlobj*, *perlmod*, et *perlbob*.

Il n'est pas inintéressant de jeter un coup d'oeil aux modules `Tie::SecureHash` et `Class::Data::Inheritable` que l'on peut trouver sur CPAN.

11.7 AUTEUR ET COPYRIGHT

Copyright (c) 1999 Tom Christiansen. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

11.8 REMERCIEMENTS

Russ Allbery, Jon Orwant, Randy Ray, Larry Rosler, Nat Torkington, and Stephen Warren all contributed suggestions and corrections to this piece. Thanks especially to Damian Conway for his ideas and feedback, and without whose indirect prodding I might never have taken the time to show others how much Perl has to offer in the way of objects once you start thinking outside the tiny little box that today's "popular" object-oriented languages enforce.

11.9 HISTORIQUE

Dernière édition : Sun Feb 4 20:50:28 EST 2001

11.10 TRADUCTION

11.10.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

11.10.2 Traducteur

Traduction initiale : Jean-Pierre Vidal <jeanpierre.vidal@free.fr>. Mise à jour: Paul Gaborit <paul.gaborit@enstimac.fr>.

11.10.3 Relecture

Aucune pour le moment.

Chapitre 12

perlbot

Collection de trucs et astuces pour Objets (the BOT)

12.1 DESCRIPTION

L'ensemble suivant d'astuces et d'indications a pour intention d'attiser la curiosité sur des sujets tels que les variables d'instance et les mécanismes des relations entre objets et classes. Le lecteur est encouragé à consulter des manuels appropriés pour toute discussion sur les définitions et méthodologies de l'Orienté Objet. Ce document n'a pas pour but d'être un cours sur la programmation orientée objet ou un guide détaillé sur les caractéristiques orientées objet de Perl ni être interprété comme un manuel. Si vous cherchez des tutoriels, regardez du côté de *perlboot*, *perltoot* et *perltooc*.

La devise de Perl reste : il existe plus d'une manière de le faire.

12.2 SÉRIE DE CONSEILS OO

1. N'essayez pas de contrôler le type de `$self`. Cela ne marchera pas si la classe est héritée, lorsque le type de `$self` est valide mais que son package n'est pas ce dont à quoi vous vous attendez. Voir règle 5.
2. Si une syntaxe orientée objet (OO) ou objet indirect (IO) à été utilisée, alors l'objet a probablement un type correct, inutile de devenir paranoïaque pour cela. Perl n'est de toute façon pas paranoïaque, lui. Si des utilisateurs de l'objet corrompent la syntaxe OO ou IO, ils savent probablement ce qu'ils font et vous devriez les laisser faire. Voir règle 1.
3. Utilisez la forme à deux arguments de `bless()`. Laissez une classe dérivée utiliser votre constructeur. Voir HÉRITAGE D'UN CONSTRUCTEUR.
4. La classe dérivée peut connaître certaines choses concernant sa classe de base immédiate, la classe de base ne peut rien connaître à propos de la classe dérivée.
5. Ne soyez pas impulsifs avec les héritages. Une relation « d'utilisation », de « contenant » ou « délégation » (du moins, ce genre de mélange) est souvent plus appropriée. Voir RELATIONS ENTRE OBJETS, UTILISATION DE RELATIONS SDBM, et DÉLÉGATION (§12.12).
6. L'objet est l'espace de nom. Y faire des packages globaux les rend accessibles par l'objet. Voilà qui éclaircit le flou du package propriétaire des objets. Voir OBJET ET CONTEXTE DE CLASSE.
7. La syntaxe IO est certainement moins compliquée mais incline aux ambiguïtés qui peuvent causer des difficultés à trouver les bogues. Permettez l'utilisation de la bonne syntaxe OO même si cela ne vous dit pas trop.
8. N'utilisez pas la syntaxe d'appel de fonction, sur une méthode. Vous rencontrerez des difficultés un de ces jours. Un utilisateur pourrait déplacer cette méthode dans une classe de base et votre code serait endommagé. En plus de cela, vous attiserez la paranoïa de la règle 2.
9. Ne présumez pas connaître le package propriétaire d'une méthode. Vous rendrez difficile pour un utilisateur de remplacer cette méthode. Voir PRÉVOIR LA RÉUTILISATION DU CODE.

12.3 VARIABLES D'INSTANCE

Un tableau anonyme ou une table de hachage anonyme peuvent être utilisés pour conserver des variables d'instance. Des paramètres nommés sont également expliqués.

```
package Foo;

sub new {
    my $type = shift;
    my %params = @_;
    my $self = {};
    $self->{'High'} = $params{'High'};
    $self->{'Low'} = $params{'Low'};
    bless $self, $type;
}

package Bar;

sub new {
    my $type = shift;
    my %params = @_;
    my $self = [];
    $self->[0] = $params{'Left'};
    $self->[1] = $params{'Right'};
    bless $self, $type;
}

package main;

$a = Foo->new( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";

$b = Bar->new( 'Left' => 78, 'Right' => 40 );
print "Left=$b->[0]\n";
print "Right=$b->[1]\n";
```

12.4 VARIABLES D'INSTANCE SCALAIRES

Une variable scalaire anonyme peut être utilisée quand seulement une variable d'instance est nécessaire.

```
package Foo;

sub new {
    my $type = shift;
    my $self;
    $self = shift;
    bless \$self, $type;
}

package main;

$a = Foo->new( 42 );
print "a=$$a\n";
```

12.5 HÉRITAGE DE VARIABLES D'INSTANCE

Cet exemple illustre comment on peut hériter des variables d'instance à partir d'une classe de base afin de permettre l'inclusion dans une nouvelle classe. Ceci nécessite l'appel du constructeur d'une classe de base et l'addition d'une variable au nouvel objet.

```
package Bar;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'buz'} = 42;
    bless $self, $type;
}

package Foo;
@ISA = qw( Bar );

sub new {
    my $type = shift;
    my $self = Bar->new;
    $self->{'biz'} = 11;
    bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

12.6 RELATIONS ENTRE OBJETS

Ce qui suit illustre comment on peut mettre en effet les relations de « contenant » et « d'utilisation » entre objets.

```
package Bar;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'buz'} = 42;
    bless $self, $type;
}

package Foo;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'Bar'} = Bar->new;
    $self->{'biz'} = 11;
    bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

12.7 REMPLACER DES MÉTHODES DE CLASSES DE BASE

L'exemple suivant illustre comment remplacer une méthode de classe de base puis appeler la méthode contournée. La **SUPER** pseudo-classe permet au programmeur d'appeler une classe contournée sans connaître vraiment où cette méthode est définie.

```

package Buz;
sub goo { print "here's the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }

package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
    my $type = shift;
    bless [], $type;
}
sub grr { print "grumble\n" }
sub goo {
    my $self = shift;
    $self->SUPER::goo();
}
sub mumble {
    my $self = shift;
    $self->SUPER::mumble();
}
sub google {
    my $self = shift;
    $self->SUPER::google();
}

package main;

$foo = Foo->new;
$foo->mumble;
$foo->grr;
$foo->goo;
$foo->google;

```

Notez bien que **SUPER** fait référence aux superclasses du paquetage courant (**FOO**) et non aux superclasses de **\$self**.

12.8 UTILISATION DE RELATIONS SDBM

Cet exemple décrit une interface pour la classe SDBM. Ceci crée une relation « d'utilisation » entre la classe SDBM et la nouvelle classe Mydbm.

```

package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw( Tie::Hash );

```

```

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'dbm' => $ref}, $type;
}
sub FETCH {
    my $self = shift;
    my $ref = $self->{'dbm'};
    $ref->FETCH(@_);
}
sub STORE {
    my $self = shift;
    if (defined $_[0]){
        my $ref = $self->{'dbm'};
        $ref->STORE(@_);
    } else {
        die "Cannot STORE an undefined key in Mydbm\n";
    }
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";

tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
$bar{'Cathy'} = 456;
print "bar-Cathy = $bar{'Cathy'}\n";

```

12.9 PRÉVOIR LA RÉUTILISATION DU CODE

L'avantage des langages OO est la facilité avec laquelle l'ancien code peut utiliser le nouveau code. L'exemple suivant illustre d'abord comment on peut empêcher la réutilisation d'un code puis comment promouvoir la réutilisation du code.

Le premier exemple illustre une classe qui utilise un appel avec une syntaxe complète, d'une méthode afin d'accéder à la méthode privée BAZ(). Le second exemple démontrera qu'il est impossible de remplacer la méthode BAZ().

```

package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package main;

$a = FOO->new;
$a->bar;

```

À présent nous essayons de remplacer la méthode BAZ(). Nous souhaiterions que FOO::bar() appelle GOOP::BAZ(), mais ceci ne peut pas se faire car FOO::bar() appelle explicitement FOO::private::BAZ().

```

package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );
sub new {
    my $type = shift;
    bless {}, $type;
}

sub BAZ {
    print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;

```

Afin de créer un code réutilisable, nous devons modifier la classe FOO, en écrasant la classe FOO::private. L'exemple suivant présente une classe FOO réutilisable qui permet à la méthode GOOP::BAZ() d'être utilisée à la place de FOO::BAZ().

```

package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->BAZ;
}

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

```

```

sub new {
    my $type = shift;
    bless {}, $type;
}
sub BAZ {
    print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;

```

12.10 OBJET ET CONTEXTE DE CLASSE

Utilisez l'objet afin de résoudre les problèmes de package et de contexte de classe. Tout ce dont une méthode a besoin doit être disponible par le biais de l'objet ou être transmis comme paramètre à la méthode.

Une classe aura parfois des données statiques ou globales qui devront être utilisées par les méthodes. Une classe dérivée peut vouloir remplacer ces données par de nouvelles. Lorsque ceci arrive, la classe de base peut ne pas savoir comment trouver la nouvelle copie de la donnée.

Ce problème peut être résolu en utilisant l'objet pour définir le contexte de la méthode. Laissez la méthode chercher dans l'objet afin de trouver une référence à la donnée. L'autre alternative est d'obliger la méthode d'aller à la chasse à la donnée (« est-ce dans ma classe ou dans une classe dérivée ? Quelle classe dérivée ? »), mais ceci peut être gênant et facilitera le piratage. Il est préférable de laisser l'objet indiquer à la méthode où la donnée est située.

```

package Bar;

%fizzle = ( 'Password' => 'XYZZY' );

sub new {
    my $type = shift;
    my $self = {};
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}

sub enter {
    my $self = shift;

    # Ne cherchez pas à deviner si on devrait utiliser %Bar::fizzle
    # ou %Foo::fizzle. L'objet sait déjà lequel
    # on doit utiliser, donc il n'y a qu'à demander.
    #
    my $fizzle = $self->{'fizzle'};

    print "The word is ", $fizzle->{'Password'}, "\n";
}

package Foo;
@ISA = qw( Bar );

%fizzle = ( 'Password' => 'Rumple' );

sub new {
    my $type = shift;
    my $self = Bar->new;
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}

package main;

$a = Bar->new;
$b = Foo->new;
$a->enter;
$b->enter;

```

12.11 HÉRITAGE D'UN CONSTRUCTEUR

Un constructeur héritable doit utiliser le deuxième forme de `bless()` qui permet de lier directement dans une classe spécifique. Notez dans cet exemple que l'objet sera un `BAR` et non un `FOO` bien que le constructeur soit dans la classe `FOO`.

```
package FOO;

sub new {
    my $type = shift;
    my $self = {};
    bless $self, $type;
}

sub baz {
    print "in FOO::baz()\n";
}

package BAR;
@ISA = qw(FOO);

sub baz {
    print "in BAR::baz()\n";
}

package main;

$a = BAR->new;
$a->baz;
```

12.12 DÉLÉGATION

Quelques classes, comme `SDBM_File`, ne peuvent pas être sous-classées correctement car elles créent des objets externes. Ce genre de classe peut être prolongée avec quelques techniques comme la relation « d'utilisation » mentionnée plus haut ou par délégation.

L'exemple suivant illustre une délégation utilisant une fonction `AUTOLOAD()` afin d'accomplir un renvoi de message. Ceci permettra à l'objet `Mydbm` de se conduire exactement comme un objet `SDBM_File`.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw(Tie::Hash);

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'delegate' => $ref};
}

sub AUTOLOAD {
    my $self = shift;

    # L'interpréteur Perl place le nom
    # du message dans une variable appelée $AUTOLOAD.

    # Un message de DESTRUCTION (DESTROY) ne doit jamais être exporté.
    return if $AUTOLOAD =~ /::DESTROY$/;
```



```
# Enlève le nom du package.
$AUTOLOAD =~ s/^Mydbm:./;

# Passe le message au délégué.
$self->{'delegate'}->$AUTOLOAD(@_);
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";
```

12.13 VOIR AUSSI

L<perlboot>, L<perltoot> et L<perltooc>.

12.14 TRADUCTION

12.14.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

12.14.2 Traducteurs

Sébastien Joncheray <info@perl-gratuit.com>. Mis à jour Paul Gaborit (Paul.Gaborit @ enstimac.fr).

12.14.3 Relecture

Simon Washbrook <swashbrook@compuserve.com>

Chapitre 13

perlstyle

Comment (bien) écrire du Perl

13.1 DESCRIPTION

Chaque programmeur aura, bien entendu, ses propres préférences dans la manière d'écrire, mais voici quelques règles générales qui rendront vos programmes plus faciles à lire, à comprendre et à mettre à jour.

La chose la plus importante est de toujours lancer vos programmes avec le paramètre `-w`. Si vous en avez vraiment besoin, vous avez la possibilité de l'enlever explicitement pour des portions de vos programmes à l'aide du pragma `no warnings` ou de la variable `$^W`. Vous devriez aussi toujours utiliser `use strict` ou au moins savoir pourquoi vous ne le faites pas. Les pragmas `use sigtrap` et même `use diagnostics` pourront s'avérer utiles.

Pour ce qui est de l'esthétique du code, la seule chose à laquelle Larry tient vraiment, c'est que les accolades fermantes d'un BLOC multiligne soient alignées avec le mot-clé qui marque le début du bloc. Après ça, il a d'autres conseils qui ne sont pas aussi forts :

- Indentation de 4 colonnes.
- Les accolades ouvrantes sont sur la même ligne que le mot-clé, si possible, sinon, qu'elles soient alignées avec lui.
- Un espace avant l'accolade ouvrante d'un BLOC multiligne.
- Un BLOC d'une ligne peut être mis sur la même ligne que le mot-clé, y compris les accolades.
- Pas d'espace avant le point-virgule.
- Point-virgule omis dans les BLOCS d'une seule ligne.
- Des espaces autour des opérateurs.
- Des espaces autour d'un indice « complexe » (entre crochets).
- Des lignes vides entre les parties qui font des choses différentes.
- Des else bien visibles.
- Pas d'espace entre le nom de fonction et la parenthèse ouvrante.
- Un espace après chaque virgule.
- Couper les lignes trop longues après un opérateur (sauf `and` et `or`).
- Un espace après la dernière parenthèse fermante sur la ligne courante.
- Aligner les items correspondants verticalement.
- Omettre la ponctuation redondante tant que la lisibilité n'en est pas affectée.

Larry a ses propres raisons pour chacune de ces choses, mais il sait bien que tout le monde ne pense pas comme lui.

Voici quelques autres choses plus concrètes auxquelles il faut penser :

- Ce n'est pas parce que vous *POUVEZ* faire quelque chose d'une façon particulière que vous *DEVEZ* le faire de cette manière. Perl a été conçu pour vous offrir plusieurs possibilités de faire une chose précise, alors, pensez à prendre la plus compréhensible. Par exemple :

```
open(F00,$foo) || die "J'arrive pas à ouvrir $foo: $!";
```

est mieux que :

```
die "J'arrive pas a ouvrir $foo: $!" unless open(F00,$foo);
```

et ce, parce que la deuxième méthode ne met pas en avant l'instruction intéressante. D'un autre côté :

```
print "Début de l'analyse\n" if $verbose;
```

est mieux que :

```
$verbose && print "Début de l'analyse\n";
```

car l'intérêt n'est pas de savoir si l'utilisateur a tapé `-v` ou non.

D'une manière similaire, ce n'est pas parce qu'un opérateur suppose des arguments par défaut qu'il faut que vous utilisiez ces arguments. Les réglages par défaut sont faits pour les programmeurs systèmes paresseux qui écrivent un programme pour une seule utilisation. Si vous voulez que votre programme soit lisible, mettez les arguments.

Dans le même ordre d'idée, ce n'est pas parce que les parenthèses *peuvent* être omises qu'il ne faut pas en mettre :

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

Quand vous avez un doute, mettez des parenthèses. Au moins, ça permettra aux pauvres gars de s'en remettre à la touche `%` sous `vi`.

Même si vous êtes sûr de vous, pensez à la santé mentale de la personne qui aura à mettre à jour le code après vous et qui mettra très certainement les parenthèses au mauvais endroit.

- Ne faites pas de contorsions impossibles pour réussir à sortir d'une boucle, Perl fournit l'opérateur `last` qui vous permet de sortir. Faites le juste dépasser pour le rendre plus visible (« outdent » en anglais).

LINE:

```
for (;;) {
    instructions;
    last LINE if $truc;
    next LINE if /^#/;
    instructions;
}
```

- N'ayez pas peur d'utiliser des labels de boucle. Ils sont là pour rendre le code plus lisible autant que pour permettre de sauter plusieurs niveaux de boucles. Référez-vous à l'exemple précédent.
- Évitez d'utiliser `grep()` (ou `map()`) ou des « backticks » (```) dans un contexte vide, c'est-à-dire, quand vous ne récupérez pas les valeurs qu'elles retournent. Ces fonctions retournent toujours des valeurs alors, utilisez-les. Sinon, à la place, utilisez une boucle `foreach()` ou la fonction `system()`.
- Pour conserver la portabilité, quand vous utilisez des fonctionnalités qui ne seront peut-être pas implémentées sur toutes les machines, testez les instructions dans un `eval` pour voir si elles échouent. Si vous savez à partir de quelle version et quel niveau de patch la fonctionnalité a été implémentée, vous pouvez tester `$]` (`$PERL_VERSION` en Anglais) pour voir si elle est présente. Le module `Config` vous permettra aussi de savoir quelles options ont été retenues par le programme `Configure` quand Perl a été installé.
- Choisissez des identificateurs ayant un sens mnémonique. Si vous n'arrivez pas à vous rappeler à quoi ils correspondent, vous avez un problème.
- Bien que les petits identificateurs comme `$nbmots` sont compréhensibles, les identificateurs longs sont plus lisibles si les mots sont séparés par des underscores. Il est plus facile de lire `$un_nom_de_variable` que `$UnNomDeVariable`, surtout pour ceux qui ne parlent pas très bien la langue dans laquelle le programme a été écrit. C'est aussi une règle simple qui marche aussi avec `UN_NOM_DE_CONSTANTE`.

Les noms de paquetages font parfois exception à la règle. Perl réserve de façon informelle des noms de modules en minuscules pour des modules « pragma » tels `integer` ou `strict`. Les autres modules devraient commencer avec une majuscule et utiliser une casse variée, mais ne mettez pas d'underscores à cause des limitations dans les noms des modules sur certains systèmes de fichiers primitifs qui ne permettent que quelques caractères.

- Vous pouvez trouver utile de laisser la casse indiquer la visibilité ou la nature d'une variable. Par exemple :

```
$TOUT_EN_MAJUSCULES    les constantes uniquement (attention
                        aux collisions avec les variables internes
                        de Perl !)

$Quelques_majuscules  variables globales/statiques
$pas_de_majuscule     internes à une fonction (my () ou local())
```

Les noms de fonctions et de méthodes semblent mieux marcher quand elles sont en minuscule. Ex : `$obj->as_string()`.

Vous pouvez utiliser un underscore au début de la variable pour indiquer qu'elle ne doit pas être utilisée hors du paquetage qui la définit.

- Si vous avez une expression régulière de la mort, utilisez le modificateur `/x` et ajoutez un peu d'espaces pour que cela ressemble un peu plus à quelque chose. N'utilisez pas de slash comme délimiteurs quand votre `regex` contient des slash ou des antislash.
- Utilisez les nouveaux opérateurs `and` et `or` pour éviter d'avoir à mettre trop de parenthèses dans les listes d'opérations et pour réduire l'utilisation des opérateurs tels `&&` et `||`. Appelez vos routines comme s'il s'agissait de fonctions ou d'opérateurs de listes pour éviter le surplus de parenthèses et de « & ».
- Utilisez des `here` (opérateur `<<`) plutôt que des instructions `print()` répétés.
- Alignez ce qui correspond verticalement, spécialement si c'est trop long pour tenir sur une seule ligne.

```
$IDX = $ST_MTIME;
$IDX = $ST_ETIME   if $opt_u;
$IDX = $ST_CTIME   if $opt_c;
```

```

$IDX = $ST_SIZE      if $opt_s;
mkdir $tmpdir, 0700 or die "je peux pas faire mkdir $tmpdir: $!";
chdir($tmpdir)      or die "je peux pas faire chdir $tmpdir: $!";
mkdir 'tmp', 0777 or die "je peux pas faire mkdir $tmpdir/tmp: $!";

```

- Vérifiez toujours la valeur retournée par un appel système. Les meilleurs messages d'erreur sont ceux qui sont dirigés sur STDERR et qui fournissent : le programme en cause, l'appel système qui a échoué avec ses arguments et (TRÈS IMPORTANT) le message d'erreur système indiquant la cause de l'échec. Voici un exemple simple, mais suffisant :

```

opendir(D, $dir)      or die "je peux pas faire opendir $dir: $!";

```
- Alignez vos translittérations quand cela a un sens :

```

tr [abc]
  [xyz];

```
- Pensez à la réutilisation du code. Pourquoi perdre de l'énergie en produisant un code jetable alors que vous aurez certainement à refaire quelque chose de similaire dans quelque temps ? Pensez à généraliser votre code. Pensez à écrire un module ou une classe. Pensez à faire tourner votre code proprement avec `use strict` et `use warnings` (ou `-w`) activés. Pensez à distribuer votre code. Pensez à changer votre regard sur le monde. Pensez à... oh, non, oubliez.
- Pensez à documenter votre code en utilisant le formatage Pod de manière cohérente. Voici quelques conventions courantes :
 - Utilisez `C<>` pour les noms de fonctions, de variables et de modules (et plus généralement pour tout ce qui peut être considéré comme du code tel que les filehandles ou des valeurs spécifiques). Remarquez qu'un nom de fonction est considéré comme plus lisible si il est suivi de parenthèses comme pour `fonction()`.
 - Utilisez `B<>` pour les commandes comme **cat** ou **grep**.
 - Utilisez `F<>` ou `C<>` pour les noms de fichiers. `F<>` devrait être utilisé systématiquement pour les noms de fichiers mais comme de nombreux traducteurs Pod le traduisent par de l'italique, les chemins Unix et Windows avec des `/` et des `\` peuvent être moins lisibles et plus présentables par un `C<>`.
- Soyez cohérents.
- Soyez gentils.

13.2 TRADUCTION

13.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

13.2.2 Traducteur

Traduction initiale : Matthieu Arnold <arn@multimania.com>

Mise à jour : Paul Gaborit <Paul.Gaborit @ enstimac.fr>

13.2.3 Relecture

Gérard Delafond.

Chapitre 14

perlcheat

Anti-sèche Perl 5

14.1 DESCRIPTION

Cette anti-sèche est une carte de référence pensée pour les programmeurs débutants en Perl. Tout n'y est pas mais plus de 195 fonctionnalités y sont déjà présentées.

14.1.1 L'anti-sèche

CONTEXTES SIGILS	TABLEAUX	HACHAGES
vide \$scalaire	tout: @tableau	%hachage
scalaire @tableau	tranche: @tableau[0, 2]	@hachage{'a', 'b'}
liste %hachage	élément: \$tableau[0]	\$hachage{'a'}
&sub		
*glob	VALEURS SCALAIRES	
	nombre, chaîne, référence, glob, undef	
RÉFÉRENCES		
\	références	\$\$foo[1] equiv. \$foo->[1]
\$_%&*	déréférence	\$\$foo{bar} equiv. \$foo->{bar}
[]	ref.tableau anon.	\$\$\$foo[1][2] equiv. \$foo->[1]->[2]
{}	ref.hachage anon.	\$\$\$foo[1][2] equiv. \$foo->[1][2]
\()	liste de références	
	NOMBRES ou CHAÎNES	LIENS
PRIORITÉ DES OPÉRATEURS =	=	perl.plover.com
->	+	search.cpan.org
++ --	== !=	eq ne cpan.org
**	< > <= >=	lt gt le ge pm.org
! ~ \ u+ u-	<=>	cmp tpj.com
=~ !~		perldoc.com
* / % x	SYNTAXE	
+ - .	for (LISTE) { }, for (a;b;c) { }	
<< >>	while () { }, until () { }	
uops nommé	if () { } elsif () { } else { }	
< > <= >= lt gt le ge	unless () { } elsif () { } else { }	
== != <=> eq ne cmp	for équivalent à foreach (TOUJOURS)	
&		
^	META-CARACTÈRES DE REGX	MODIF. DE REGEX
&&	^	début chaîne /i insens. casse
	\$	fin chaîne (avant \n) /m ^\$ multilignes
.. ...	+	un ou plus /s . inclut \n
?:	*	zéro ou plus /x ignore blancs
= += -= *= etc.	?	zéro ou un /g global
, =>	{3,7}	nb de répétitions /o compil. unique

ops de liste	()	capture	
not	(?:)	sans capture	CLASSES DE CAR. REGEX
and	[]	classe caractères	. == [^\n]
or xor		choix	\s == les espaces
	\b	limite de mot	\w == caract. de mots
	\z	fin de chaîne	\d == chiffres
UTILISEZ			inv. par \S, \W et \D
use strict;		N'UTILISEZ PAS	
use warnings;		"\$foo"	LIENS
my \$var;		\$\$nom_variable	perl.com
open() or die \$!;		'\$saisie_utilisateur'	use.perl.org
use Modules;		/\$saisie_utilisateur/	perl.apache.org

FONCTIONS RETOURNANT DES LISTES

stat	localtime	caller	VARIABLES SPÉCIALES
0 dev	0 seconde	0 package	\$_ variable par défaut
1 ino	1 minute	1 nomfichier	\$0 nom du programme
2 mode	2 heure	2 ligne	\$/ séparateur d'entrée
3 nlink	3 jour	3 subroutine	\$\ séparateur de sortie
4 uid	4 mois-1	4 avec args	\$ autoflush
5 gid	5 annee-1900	5 wantarray	\$! erreur appel sys/lib
6 rdev	6 j/semaine	6 eval texte	\$@ erreur eval
7 size	7 j/anne	7 is_require	\$\$ ID du processus
8 atime	8 heure été	8 hints	\$. numero ligne
9 mtime		9 bitmask	@ARGV args ligne commande
10 ctime	just use		@INC chemin inclusion
11 blkisz	POSIX::	3..9 only	@_ args subroutine
12 blkisz	strftime!	with EXPR	%ENV environnement

14.2 REMERCIEMENTS

La première version de document est paru sur Perl Monks. Plusieurs personnes y ont fait de bonnes suggestions. Merci aux "Perl Monks".

Un remerciement spécial à Damian Conway qui, en plus de proposer des changements importants, a pris le temps de compter le nombre de fonctionnalités présentées et d'en faire une version Perl 6 pour montrer que Perl restera Perl.

14.3 AUTEUR

Juere Waalboer <#####@juere.nl>, avec l'aide de nombreux "Perl Monks".

14.4 VOIR AUSSI

http://perlmonks.org/?node_id=216602	le document PM original
http://perlmonks.org/?node_id=238031	la version Perl 6 de Damian Conway
http://juere.nl/site.plp/perlcheat	le site de l'anti-sèche

14.5 TRADUCTION

14.5.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

14.5.2 Traducteur

Traduction, parfois approximative, par Paul Gaborit (Paul.Gaborit at enstimac.fr).

14.5.3 Relecture

Personne pour l'instant.

Chapitre 15

perltrap

Les pièges de Perl pour l'imprudent

15.1 DESCRIPTION

Le plus gros piège est d'oublier d'utiliser la directive `use warnings` ou l'option `-w` ; voir *perllexwarn* et *perlrun*. Le deuxième plus gros piège est de ne pas rendre la totalité de votre programme exécutable sous `use strict`. Le troisième plus grand piège est de ne pas lire la liste des changements effectués dans cette version de Perl ; voir *perldelta*.

15.1.1 Pièges de `awk`

Les utilisateurs avertis de `awk` devraient se soucier particulièrement de ce qui suit :

- Un programme Perl est exécuté une seule fois et non pour chacune des lignes d'entrée. Vous pouvez obtenir une boucle implicite en utilisant les options `-n` et `-p`.
- Le module `English`, chargé par `use English` ; vous permet de vous référer aux variables spéciales (comme `$/`) par des noms (comme `$RS`), comme si elles étaient en `awk` ; voir *perlvar* pour plus de détails.
- Le point-virgule est requis après toute instruction simple en Perl (sauf à la fin d'un bloc). La fin de ligne n'est pas un délimiteur d'instruction.
- Les accolades sont requises pour les `ifs` et les `whiles`.
- Les variables commencent par "\$", "@" ou "%" en Perl.
- Les tableaux sont indexés à partir de 0. De la même manière, les positions renvoyées par `substr()` et `index()` dans les chaînes sont comptées à partir de 0.
- Vous devez décider si votre tableau a pour indices des nombres ou des chaînes.
- Les hachages ne sont pas créés par une simple référence.
- Vous devez décider si une comparaison porte sur des chaînes ou des nombres.
- La lecture d'une entrée ne la découpe pas pour vous. Vous devez la transformer en un tableau vous-même. Et l'opérateur `split()` a des arguments différents qu'en `awk`.
- La ligne courante en entrée est normalement dans `$_`, pas dans `$0`. Elle n'a en général pas encore perdu sa fin de ligne (`$0` est le nom du programme exécuté). Voir *perlvar*.
- `$digit` ne se réfère pas à des champs – il se réfère aux sous-chaînes détectées par le dernier motif de correspondance.
- L'instruction `print()` n'ajoute pas de séparateurs de champs et d'enregistrements à moins que vous ne fixiez `$,` et `$\`. Vous pouvez fixer `$OFS` et `$ORS` si vous utilisez le module `English`.
- Vous devez ouvrir vos fichiers avant d'y écrire.
- L'opérateur d'intervalle est `..`, et pas la virgule. L'opérateur virgule fonctionne comme en C.
- L'opérateur de correspondance est `=~`, pas `~~` ("`~`" est l'opérateur de complément à 1, comme en C).
- L'opérateur d'exponentiation est `**`, pas `^`. "`^`" est l'opérateur XOR, comme en C (vous savez, on pourrait finir par avoir la sensation que `awk` est intrinsèquement incompatible avec C).
- L'opérateur de concaténation est `.`, pas la chaîne vide (utiliser la chaîne vide rendrait `/pat/ /pat/` impossible à analyser lexicalement, car le troisième signe de division serait interprété comme un opérateur de division – le détecteur de mots-clés est en fait légèrement sensible au contexte pour les opérateurs tels que `/`, `?`, et `>`. Et en fait, `.` lui-même peut être le début d'un nombre).
- Les mots-clés `next`, `exit`, et `continue` fonctionnent différemment.

– Les variables suivantes fonctionnent différemment :

```

Awk      Perl
ARGC     scalar @ARGV (à comparer avec $#ARGV)
ARGV[0]  $0
FILENAME $ARGV
FNR      $. - quelque chose
FS       (ce que vous voulez)
NF       $#Fld, ou quelque chose comme ça
NR       $.
OFMT     $#
OFS      $,
ORS      $\
RLENGTH  length($&)
RS       $/
RSTART   length($')
SUBSEP   $;

```

- Vous ne pouvez pas mettre un motif dans \$RS, seulement une chaîne.
- En cas de doute, faites passer la construction **awk** dans **a2p** et voyez ce que cela donne.

15.1.2 Pièges de C

Les programmeurs C et C++ devraient prendre note de ce qui suit :

- Les accolades sont requises pour les `ifs` et les `whiles`.
- Vous devez utiliser `elsif` à la place de `else if`.
- Les mots-clés `break` et `continue` de C deviennent respectivement `last` et `next` en Perl. Contrairement au C, ceux-ci ne fonctionnent *pas* à l'intérieur d'une structure `do { } while`. Voir Contrôle de Boucle in *perlsyn*.
- Il n'y a pas d'instruction `switch` (mais il est aisé d'en construire une à la volée, voir BLOCs de Base et Instruction Switch in *perlsyn*).
- Les variables commencent par "\$", "@" ou "%" en Perl.
- Les commentaires commencent par "#", pas par "/" ou "//". Perl peut interpréter un commentaire C/C++ comme un opérateur de division, un expression rationnelle non terminée ou comme l'opérateur "défini-ou".
- Vous ne pouvez pas récupérer l'adresse de quelque chose, bien que l'opérateur barre oblique inverse de Perl vous permette de faire quelque chose d'assez proche : créer une référence.
- ARGV doit être en majuscules. \$ARGV[0] est l'argv[1] du C, et argv[0] atterrit dans \$0.
- Les appels systèmes tels que `link()`, `unlink()`, `rename()`, etc. renvoient autre chose que zéro en cas de succès. (En revanche, `system()` retourne un zéro en cas de succès.)
- Les handlers de signaux manipulent des noms de signaux, pas des nombres. Utilisez `kill -1` pour déterminer leurs noms sur votre système.

15.1.3 Pièges de sed

Les programmeurs `sed` expérimentés devraient prendre note de ce qui suit :

- Un programme Perl est exécuté une seule fois et non pour chacune des lignes d'entrée. Vous pouvez obtenir une boucle implicite en utilisant les options `-n` et `-p`.
- Les références arrières dans les substitutions utilisent "\$" à la place de "\\".
- Les métacaractères d'expressions rationnelles ("(", ")", et "|") ne sont pas précédés de barres obliques inverses.
- L'opérateur d'intervalle est `..`, à la place de la virgule.

15.1.4 Pièges du shell

les programmeurs shell dégourdis devraient prendre note de ce qui suit :

- L'opérateur accent grave effectue une interpolation de variable sans se soucier de la présence ou non d'apostrophes dans la commande.
- L'opérateur accent grave ne fait pas de traduction de la valeur de retour, contrairement au **csh**.
- Les shells (en particulier **csh**) effectuent plusieurs niveaux de substitution sur chaque ligne de commande. Perl ne fait de substitution que dans certaines structures comme les guillemets, les accents graves, les crochets et les motifs de recherche.
- Les shells interprètent les scripts petits bouts par petits bouts. Perl compile tout le programme avant de l'exécuter (sauf les blocs BEGIN, qu'il exécute lors de la compilation).

- Les arguments sont disponibles via `@ARGV`, et pas `$1`, `$2`, etc.
- L'environnement n'est pas automatiquement mis à disposition sous forme de variables scalaires distinctes.
- Les tests en shell utilisent "=", "!=", "<", etc. pour les comparaisons de chaînes et "-eq", "-ne", "-lt", etc. pour les comparaisons numériques. En Perl, c'est exactement le contraire : on utilise `eq`, `ne`, `lt`, etc. pour les comparaisons de chaînes et `==`, `!=`, `<`, etc. pour les comparaisons numériques.

15.1.5 Pièges de Perl

Les programmeurs Perl pratiquants devraient prendre note de ce qui suit :

- Souvenez-vous que de nombreuses opérations se comportent différemment selon qu'elles sont appelées dans un contexte de liste ou dans un contexte scalaire. Voir *perldata* pour plus de détails.
- Évitez autant que possible les barewords, en particulier ceux en minuscules. Vous ne pouvez pas dire rien qu'en le regardant si un bareword est une fonction ou une chaîne. En utilisant les apostrophes pour les chaînes et des parenthèses pour les appels de fonctions, vous permettrez qu'on ne les confonde jamais.
- Vous ne pouvez pas distinguer, par simple inspection, les fonctions intégrées qui sont, par construction, des opérateurs unaires (comme `chop()` et `chdir()`) de celles qui sont des opérateurs de liste (comme `print()` et `unlink()`). (Sauf à utiliser des prototypes, les sous-programmes définis par l'utilisateur ne peuvent être **que** des opérateurs de liste, jamais des opérateurs unaires). Voir *perlop* et *perlsub*.
- Les gens ont du mal à se souvenir que certaines fonctions utilisent par défaut `$_`, ou `@ARGV`, ou autre chose, tandis que d'autres ne le font pas alors qu'on pourrait s'y attendre.
- La structure `<FH>` n'est pas le nom du handle de fichier, c'est un opérateur lisant une ligne sur ce handle. Les données lues sont affectées à `$_` seulement si la lecture du fichier est la seule condition d'une boucle `while` :


```
while (<FH>) { }
while (defined($_ = <FH>)) { }..
<FH>; # données rejetées !
```
- Souvenez-vous de ne pas utiliser `=` lorsque vous avez besoin de `=~` ; ces deux structures sont très différentes :


```
$x = /foo/;
$x =~ /foo/;
```
- La structure `do { }` n'est pas une véritable boucle sur laquelle vous pourriez utiliser les instructions de contrôle de boucle.
- Utilisez `my()` pour les variables locales chaque fois que vous pouvez vous en satisfaire (mais voyez *perlform* pour les cas où vous ne le pouvez pas). L'utilisation de `local()` donne vraiment une valeur locale à une variable globale, ce qui vous met à la merci d'effets secondaires de portée dynamique imprévus.
- Si vous localisez une variable exportée dans un module, sa valeur exportée *value* ne changera pas. Le nom local devient un alias pour une nouvelle valeur, mais le nom externe est toujours un alias de l'original.

15.1.6 Pièges entre Perl4 et Perl5

Les programmeurs Perl4 pratiquants devraient prendre note des pièges suivants, spécifiques au passage entre Perl4 et Perl5.

Ils sont crûment commandés par la liste suivante :

Pièges liés aux corrections de bugs, aux désapprobations et aux abandons

Tout ce qui a été corrigé en tant que bug de perl4, supprimé ou désapprouvé en tant que caractéristique de perl4, avec l'intention d'encourager l'usage d'une autre caractéristique de perl5.

Pièges de l'analyse syntaxique

Pièges qui semblent provenir du nouvel analyseur.

Pièges numériques

Pièges liés aux opérateurs numériques ou mathématiques.

Pièges des types généraux de données

Pièges impliquant les types de données standards de perl.

Pièges du contexte - contexte scalaire et de liste

Pièges liés au contexte dans les instructions et déclarations scalaires ou de liste.

Pièges de la précedence (les priorités)

Pièges liés à la précedence lors de l'analyse, de l'évaluation et de l'exécution du code.

Pièges des expressions rationnelles générales lors de l'utilisation de `s///`, etc.

Pièges liés à l'utilisation de la reconnaissance de motifs.

Pièges des sous-programmes, des signaux, des tris

Pièges liés à l'utilisation des signaux et des handlers de signaux, aux sous-programmes généraux, et aux tris, ainsi qu'aux sous-programmes de tri.

Pièges du système d'exploitation

Pièges spécifiques au système d'exploitation.

Pièges de DBM

Pièges spécifiques à l'utilisation de `dbmopen()`, et aux implémentations particulières de `dbm`.

Pièges non classés

Tout le reste.

Si vous trouvez un exemple de piège de conversion qui ne soit pas listé ici, soumettez-le à [<perlbug@perl.org>](mailto:perlbug@perl.org) en vue de son inclusion. Notez aussi qu'un certain nombre d'entre eux peuvent être détectés par le pragma `use warnings` ou par l'option `-w`.

15.1.7 Pièges liés aux corrections de bugs, aux désapprobations et aux abandons

Tout ce qui a été abandonné, désapprouvé ou corrigé comme étant un bug depuis perl4.

- Les symboles commençant par "_" ne sont plus forcément dans `main`.

Les symboles commençant par "_" ne sont plus forcément dans le paquetage `main`, sauf pour `$_` lui-même (ainsi que `@_`, etc.).

```
package test;
$_legacy = 1;
package main;
print "\$_legacy is ", $_legacy, "\n";
# perl4 affiche : $_legacy is 1
# perl5 affiche : $_legacy is
```

- Le double deux-points dans un nom de variable est un séparateur de nom de paquetage

Le double deux-points est désormais un séparateur de paquetage valide dans un nom de variable. Ainsi ceux-ci se comportent différemment en perl4 et en perl5, car le paquetage n'existe pas.

```
$a=1;$b=2;$c=3;$var=4;
print "$a::$b::$c ";
print "$var::abc::xyz\n";
# perl4 affiche : 1::2::3 4::abc::xyz
# perl5 affiche : 3
```

Étant donné que `::` est maintenant le délimiteur de paquetage préféré, on peut se demander si ceci doit être considéré comme un bug ou non (l'ancien délimiteur de paquetage, `'`, est utilisé ici).

```
$x = 10 ;
print "x=${'x}\n" ;
# perl4 affiche : x=10
# perl5 affiche : Can't find string terminator "'" anywhere before EOF
```

Vous pouvez éviter ce problème, en restant compatible avec perl4, si vous incluez toujours explicitement le nom du paquetage :

```
$x = 10 ;
print "x=${main'x}\n" ;
```

Voyez aussi les pièges de précedence, pour l'analyse de `$:`.

- Les deuxièmes et troisièmes arguments de `splice()` ont un contexte scalaire

Les deuxièmes et troisièmes arguments de `splice()` sont désormais évalués dans un contexte scalaire (comme l'indique le Camel) plutôt que dans un contexte de liste.

```
sub sub1{return(0,2) } # retourne une liste de 2 éléments
sub sub2{ return(1,2,3)} # retourne une liste de 3 éléments
@a1 = ("a","b","c","d","e");
@a2 = splice(@a1,&sub1,&sub2);
print join(' ',@a2),"\n";
# perl4 affiche : a b
# perl5 affiche : c d e
```

- Impossible de faire un `goto` vers un bloc optimisé

Vous ne pouvez pas faire un `goto` dans un bloc optimisé. Mince.

```
goto marker1;
```

- ```

for(1){
marker1:
 print "Here I is!\n";
}
perl4 affiche : Here I is!
perl5 erreur : Can't "goto" into the middle of a foreach loop

```
- Impossible d'utiliser l'espace dans un nom de variable ou comme délimiteur  
Il n'est plus légal syntaxiquement d'utiliser l'espace comme nom de variable, ou comme délimiteur pour tout type de structure entre apostrophes. Mince et remince.

```

$a = ("foo bar");
$b = q baz ;
print "a is $a, b is $b\n";
perl4 affiche : a is foo bar, b is baz
erreur de perl5 : Bareword found where operator expected

```
  - while/if BLOCK BLOCK a disparu  
La syntaxe archaïque while/if BLOC BLOC n'est plus supportée.

```

if { 1 } {
 print "True!";
}
else {
 print "False!";
}
perl4 affiche : True!
erreur de perl5 : syntax error at test.pl line 1, near "if {"

```
  - \*\* est prioritaire sur le moins unaire  
L'opérateur \*\* est désormais plus prioritaire que le moins unaire. Cela devait fonctionner ainsi selon la documentation, mais ce n'était pas le cas.

```

print -4**2,"\n";
perl4 affiche : 16
perl5 affiche : -16

```
  - foreach a changé lorsqu'il est appliqué à une liste  
La signification de foreach{} a légèrement changé lorsqu'il traverse une liste qui n'est pas un tableau. Auparavant, il fonctionnait en affectant la liste à un tableau temporaire, mais ce n'est plus le cas (pour des raisons d'efficacité). Cela veut dire que vous itérez maintenant sur les vraies valeurs de la liste, pas sur des copies de ces valeurs. Les modifications de la variable de boucle peuvent changer les valeurs originelles.

```

@list = ('ab', 'abc', 'bcd', 'def');
foreach $var (grep(/ab/,@list)){
 $var = 1;
}
print (join(':',@list));
perl4 affiche : ab:abc:bcd:def
perl5 affiche : 1:1:bcd:def

```

Pour garder la sémantique de Perl4, vous devrez affecter explicitement votre liste à un tableau temporaire puis itérer dessus. Vous pourriez par exemple devoir changer

```

foreach $var (grep(/ab/,@list)){
en
 foreach $var (@tmp = grep(/ab/,@list)){

```

Sinon changer \$var modifiera les valeurs de @list (Cela arrive le plus souvent lorsque vous utilisez \$\_ comme variable de boucle, et appelez dans celle-ci des sous-programmes qui ne localisent pas correctement \$\_).
  - split sans argument a changé de comportement  
split sans argument se comporte désormais comme split ' ' (qui ne retourne pas de champ nul initial si \$\_ commence par une espace), il se comportait habituellement comme split /\s+/ (qui le fait).

```

$_ = ' hi mom';
print join(':', split);
perl4 affiche : :hi:mom
perl5 affiche : hi:mom

```
  - comportement corrigé de l'option -e  
Perl 4 ignorait tout texte attaché à une option -e, et prenait toujours le petit bout de code dans l'argument suivant. Qui plus est, il acceptait silencieusement une option -e non suivie d'un argument. Ces deux comportements ont été supprimés.

```

perl -e'print "attached to -e"' 'print "separate arg"'

```

- ```

# perl4 affiche : separate arg
# perl5 affiche : attached to -e
perl -e
# perl4 affiche :
# perl5 meurt : No code specified for -e.

```
- `push` retourne le nombre d'éléments de la liste résultante
En Perl 4, la valeur de retour de `push` n'était pas documentée, mais c'était en fait la dernière valeur poussée sur la liste cible. En Perl 5, la valeur de retour de `push` est documentée, mais elle a changé, c'est désormais le nombre d'éléments de la liste résultante.

```

    @x = ('existing');
    print push(@x, 'first new', 'second new');
# perl4 affiche : second new
# perl5 affiche : 3

```
 - Certains messages d'erreur ont changé
Certains messages d'erreur sont différents.
 - `split` respecte les arguments des sous-programmes
En Perl 4, dans un contexte de liste, si les délimiteurs du premier argument de `split()` étaient ??, le résultat était placé dans `@_` tout en étant retourné normalement. Perl 5 respecte maintenant les arguments de vos sous-programmes.
 - Bogues supprimés
Certains bugs ont peut-être été supprimés par inadvertance. :-)

15.1.8 Pièges de l'analyse syntaxique

Pièges entre Perl4 et Perl5 ayant un rapport avec l'analyse syntaxique.

- Un espace entre `.` et `=` produit une erreur
Notez l'espace entre `.` et `=`

```

$string . = "more string";
print $string;
# perl4 affiche: more string
# perl5 affiche : syntax error at - line 1, near ". ="

```
- Meilleure analyse syntaxique en perl 5
L'analyse syntaxique est meilleure en perl 5

```

sub foo {}
&foo
print("hello, world\n");
# perl4 affiche : hello, world
# perl5 affiche : syntax error

```
- analyse syntaxique des fonctions
Règle "si ça a l'air d'une fonction, c'est une fonction".

```

print
($foo == 1) ? "is one\n" : "is zero\n";
# perl4 affiche : is zero
# perl5 avertit : "Useless use of a constant in void context" if using -w

```
- L'interpolation de la construction `#{tableau}` a changé
L'interpolation de chaîne de la structure `#{tableau}` diffère lorsque des accolades sont utilisées autour du nom.

```

@a = (1..3);
print "${#a}";
# perl4 affiche : 2
# perl5 échoue sur une erreur de syntaxe
@a = (1..3);
print "#{a}";
# perl4 affiche : {a}
# perl5 affiche : 2

```
- Perl devine si un `{` qui suit un `map` ou un `grep` débute un BLOC ou une référence à une table de hachage
Lorsque Perl rencontre `map {` (ou `grep {`), il doit deviner si l'accolade `{` débute un nouveau BLOC ou une référence à une table de hachage. Si il devine mal, il indiquera une erreur de syntaxe au niveau de l'accolade fermante `}` et de la virgule manquante (ou inattendue).
Utilisez un `+` unaire avant l'accolade `{` d'une référence à une table de hachage ou un `+` unaire appliqué à la première chose qui commence le BLOC (après l'accolade `{`) pour que perl devine correctement tout le temps. (Voir `map` in *perlfunc*.)

15.1.9 Pièges numériques

Pièges entre Perl4 et Perl5 ayant un rapport avec les opérateurs numériques, les opérandes, ou leurs sorties.

- Sortie formatée et chiffres significatifs

Sortie formatée et chiffres significatifs. En général, Perl 5 essaye d'être plus précis. Par exemple, sur une Sparc sous Solaris :

```
print 7.373504 - 0, "\n";
printf "%20.18f\n", 7.373504 - 0;
# Perl4 affiche :
7.373503999999996141
7.37350399999999614
# Perl5 affiche :
7.373504
7.37350399999999614
```

Remarquez que le premier résultat est mieux en Perl 5.

Les résultats que vous obtenez sont variables puisque vos fonctions d'affichage des flottants et vos flottants eux-même peuvent être différents.

- Détection du dépassement de l'entier signé maximale lors d'une auto-incrémentation

Cette détection a été rétablie. Auparavant on pouvait montre que l'opérateur d'auto-incrémentation ne voyait pas qu'un nombre avait dépassé la limite supérieure des entiers signés. Cela a été réglé dans la version 5.003_04. Mais soyez toujours prudent lorsque vous utilisez de grands entiers. En cas de doute :

```
use Math::BigInt;
```

- L'affectation du résultat d'un test d'égalité numérique ne fonctionne pas

L'affectation du résultat d'un test d'égalité numérique ne fonctionne pas en perl5 lorsque le test échoue (auparavant il renvoyait 0). Les tests logique retourne maintenant une chaîne vide, au lieu de 0.

```
$p = ($test == 1);
print $p, "\n";
# perl4 affiche : 0
# perl5 affiche :
```

Voir aussi Pièges des expressions rationnelles générales lors de l'utilisation de *s///*, etc. (§15.1.13) pour un autre exemple de cette nouvelle caractéristique...

- Opérateurs de chaîne bit par bit

Lorsque des opérateurs bit par bit qui peuvent travailler sur des nombres ou sur des chaînes (& | ^ ~) n'ont que des chaînes pour arguments, perl4 traite les opérandes comme des chaînes de bits dans la mesure où le programme contenait un appel à la fonction *vec()*. perl5 traite les opérandes chaînes comme des chaînes de bits (Voir Opérateurs bit à bit sur les chaînes in *perlop* pour plus de détails).

```
$fred = "10";
$barney = "12";
$betty = $fred & $barney;
print "$betty\n";
# Décommentez la ligne suivante pour changer le comportement de perl4
# ($dummy) = vec("dummy", 0, 0);
# Perl4 affiche :
8
# Perl5 affiche :
10
# Si vec() est utilisé quelque part dans le programme, les deux
# affichent :
10
```

15.1.10 Pièges des types de données généraux

Pièges entre Perl4 et Perl5 impliquant la plupart des types de données, et leur usage dans certaines expressions et/ou contextes.

- Les indices négatifs partent maintenant de la fin du tableau

Les indices de tableau négatifs sont maintenant comptés depuis la fin du tableau.

```
@a = (1, 2, 3, 4, 5);
print "The third element of the array is $a[3] also expressed as $a[-2] \n";
# perl4 affiche : The third element of the array is 4 also expressed as
# perl5 affiche : The third element of the array is 4 also expressed as 4
```

- Diminuer la valeur de \$#tableau supprime les éléments du tableau
 Désormais, diminuer la valeur de \$#array supprime réellement les éléments excédentaires du tableau, et rend impossible leur récupération.


```

@a = (a,b,c,d,e);
print "Before: ",join(',',@a);
$a = 1;
print ", After: ",join(',',@a);
$a = 3;
print ", Recovered: ",join(',',@a),"\n";
# perl4 affiche : Before: abcde, After: ab, Recovered: abcd
# perl5 affiche : Before: abcde, After: ab, Recovered: ab
    
```
- Les tables de hachage sont définies même vide
 Les hachages sont définis avant même d'être remplie


```

local($s,@a,%h);
die "scalar \$s defined" if defined($s);
die "array \@a defined" if defined(@a);
die "hash \%h defined" if defined(%h);
# perl4 affiche :
# perl5 meurt : hash %h defined
    
```

Perl génère désormais un avertissement lorsqu'il voit defined(@a) et defined(%h).
- Affectation globale de variable localisée à variable
 L'affectation globale de variable à variable échouera si la variable affectée est localisée après l'affectation


```

@a = ("This is Perl 4");
*b = *a;
local(@a);
print @b,"\n";
# perl4 affiche : This is Perl 4
# perl5 affiche :
    
```
- Affectation globale de undef
 L'affectation globale de undef n'a pas d'effet en Perl 5. En Perl 4, elle rend indéfini le scalaire associé (mais peut avoir d'autres effets secondaires, y compris des SEGV). Perl 5 émet un avertissement lorsqu'on affecte globalement undef (Notez bien que l'affectation globale de undef n'est pas la même chose que d'appliquer la fonction undef à une variable globale (undef \$foo)).


```

$foo = "bar";
*foo = undef;
print $foo;
# perl4 affiche :
# perl4 avertit : "Use of uninitialized variable" si -w
# perl5 affiche : bar
# perl5 avertit : "Undefined value assigned to typeglob" si -w
    
```
- Changements dans la négation unaire (de chaînes)
 Changements dans la négation unaire (de chaînes). Ce changement affecte à la fois la valeur de retour et ce qu'elle fait à l'incrément automatique.


```

$x = "aaa";
print ++$x, " : ";
print -$x, " : ";
print ++$x, "\n";
# perl4 affiche : aab : -0 : 1
# perl5 affiche : aab : -aab : aac
    
```
- Modifier une constante est interdit
 perl 4 vous laisse modifier les constantes :


```

$foo = "x";
&mod($foo);
for ($x = 0; $x < 3; $x++) {
    &mod("a");
}
sub mod {
    print "before: $_[0]";
    $_[0] = "m";
    print " after: $_[0]\n";
}
    
```

```

# perl4:
# before: x after: m
# before: a after: m
# before: m after: m
# before: m after: m
# Perl5:
# before: x after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a

```

- Le comportement de `defined $var` a changé

Le comportement est légèrement différent pour :

```

print "$x", defined $x
# perl 4: 1
# perl 5: <no output, $x is not called into existence>

```

- Suicide de variable

Le suicide de variable est plus cohérent sous Perl 5. Perl5 montre le même comportement pour les hachages et les scalaires, que celui montré par Perl4 uniquement pour les scalaires.

```

$aGlobal{ "aKey" } = "global value";
print "MAIN:", $aGlobal{"aKey"}, "\n";
$GlobalLevel = 0;
&test( *aGlobal );
sub test {
    local( *theArgument ) = @_;
    local( %aNewLocal ); # perl 4 != 5.0011,m
    $aNewLocal{"aKey"} = "this should never appear";
    print "SUB: ", $theArgument{"aKey"}, "\n";
    $aNewLocal{"aKey"} = "level $GlobalLevel"; # what should print
    $GlobalLevel++;
    if( $GlobalLevel<4 ) {
        &test( *aNewLocal );
    }
}
# Perl4:
# MAIN:global value
# SUB: global value
# SUB: level 0
# SUB: level 1
# SUB: level 2
# Perl5:
# MAIN:global value
# SUB: global value
# SUB: this should never appear
# SUB: this should never appear
# SUB: this should never appear

```

15.1.11 Pièges du contexte - contextes scalaires et de liste

- Les éléments des liste d'arguments des formats sont évalués dans un contexte de liste

Les éléments des listes d'arguments pour les formats sont désormais évalués dans un contexte de liste. Cela signifie que vous pouvez maintenant interpoler les valeurs de liste.

```

@fmt = ("foo","bar","baz");
format STDOUT=
@<<<<< @|||| @>>>>>
@fmt;
.
write;
# erreur de perl4 : Please use commas to separate fields in file
# perl5 affiche : foo bar baz

```

- `caller` retourne faux si il est évalué dans un contexte scalaire et qu'il n'y a pas d'appelant

La fonction `caller()` retourne maintenant la valeur faux dans un contexte scalaire s'il n'y a pas d'appelant. Cela laisse les fichiers de bibliothèque déterminer s'ils sont en train d'être demandés.


```

caller() ? (print "You rang?\n") : (print "Got a 0\n");
# erreur de perl4 : There is no caller
# perl5 affiche : Got a 0

```

- L'opérateur virgule dans un contexte scalaire donne un contexte scalaire à ses arguments

L'opérateur virgule dans un contexte scalaire est désormais garanti comme donnant un contexte scalaire à ses arguments.

```

@y= ('a', 'b', 'c');
$x = (1, 2, @y);
print "x = $x\n";
# Perl4 affiche : x = c # pense contexte de liste,
# interpole la liste
# Perl5 affiche : x = 3 # sait que le scalaire utilise la
# longueur de la liste

```

- `sprintf()` est prototypé comme `($;@)`

La fonction `sprintf()` utilise maintenant le prototype `($;@)`. Elle donne donc un contexte scalaire à son premier argument. Donc si vous lui passer un tableau, elle ne fera probablement pas ce que vous souhaitez contrairement à Perl 4 :

```

@z = ('%s%s', 'foo', 'bar');
$x = sprintf(@z);
print $x;
# perl4 affiche : foobar
# perl5 affiche : 3

```

`printf()` continue à fonctionner comme en Perl 4. Donc :

```

@z = ('%s%s', 'foo', 'bar');
printf STDOUT (@z);
# perl4 affiche : foobar
# perl5 affiche : foobar

```

15.1.12 Pièges de la précedence (les priorités)

Pièges entre Perl4 et Perl5 impliquant l'ordre de précedence.

Perl 4 a presque les mêmes règles de précedence que Perl 5 pour les opérateurs qu'ils ont en commun. Toutefois, Perl 4 semble avoir contenu des incohérences ayant rendu son comportement différent de celui qui était documenté.

- LHS vs. RHS pour tout opérateur d'affectation

LHS vs. RHS pour tout opérateur d'affectation. LHS est d'abord évalué en perl4, mais en second en perl5 ; ceci peut modifier les relations entre les effets de bord dans les sous-expressions.

```

@arr = ( 'left', 'right' );
$a{shift @arr} = shift @arr;
print join( ' ', keys %a );
# perl4 affiche : left
# perl5 affiche : right

```

- Erreurs sémantiques dues aux règles de précedence

Voici des expressions devenues des erreurs sémantiques à cause de la précedence :

```

@list = (1,2,3,4,5);
%map = ("a",1,"b",2,"c",3,"d",4);
$n = shift @list + 2; # premier élément de la liste plus 2
print "n is $n, ";
$m = keys %map + 2; # nombre d'éléments du hachage plus 2
print "m is $m\n";
# perl4 affiche : n is 3, m is 6
# erreur de perl5 et échec de la compilation

```

- Précedence identique pour les opérateurs d'affectation et l'affectation

La précedence des opérateurs d'affectation est désormais la même que celle de l'affectation. Perl 4 leur donnait par erreur la précedence de l'opérateur associé. Vous devez donc maintenant les mettre entre parenthèses dans les expressions telles que

```

/foo/ ? ($a += 2) : ($a -= 2);

```

Autrement

```

/foo/ ? $a += 2 : $a -= 2

```

serait analysé de façon erronée sous la forme

```

(/foo/ ? $a += 2 : $a) -= 2;

```

D'autre part,

```
$a += /foo/ ? 1 : 2;
```

fonctionne désormais comme un programmeur C pourrait s'y attendre.

- open peut nécessiter des parenthèse autour du filehandle

```
open FOO || die;
```

est désormais incorrect. Vous devez mettre le handle de fichier entre parenthèses. Sinon, perl5 laisse sa précedence par défaut à l'instruction :

```
open(FOO || die);
```

```
# perl4 ouvre ou meurt
```

```
# perl5 ouvre FOO, et ne mourrait que si 'FOO' était faux, c.-à-d. jamais
```

- Suppression de la priorité de \$: sur C\$: :

perl4 donne la précedence à la variable spéciale \$:, tandis que perl5 traite \$:: comme étant le paquetage principale

```
$a = "x"; print "$::a";
```

```
# perl 4 affiche : -:a
```

```
# perl 5 affiche : x
```

- Précedence des opérateurs de test sur fichiers telle que documentée

perl4 a une précedence bugguée pour les opérateurs de test de fichiers vis-à-vis des opérateurs d'affectation. Ainsi, bien que la table de précedence de perl4 laisse à penser que `-e $foo .= "q"` devrait être analysé comme `((-e $foo) .= "q")`, il l'analyse en fait comme `(-e ($foo .= "q"))`. En perl5, la précedence est telle que documentée.

```
-e $foo .= "q"
```

```
# perl4 affiche : no output
```

```
# perl5 affiche : Can't modify -e in concatenation
```

- keys, each, values sont des opérateurs unaires nommés normaux

En perl4, keys(), each() et values() étaient des opérateurs spéciaux de haute précedence qui agissaient sur un simple hachage, mais en perl5, ce sont des opérateurs unaires nommés normaux. Tels que documentés, ces opérateurs ont une précedence plus basse que les opérateurs arithmétiques et de concaténation + - .., mais les variantes de perl4 de ces opérateurs se lient en fait plus étroitement que + - .. Ainsi, pour :

```
%foo = 1..10;
```

```
print keys %foo - 1
```

```
# perl4 affiche : 4
```

```
# perl5 affiche : Type of arg 1 to keys must be hash (not subtraction)
```

Le comportement de perl4 était probablement plus utile, mais moins cohérent.

15.1.13 Pièges des expressions rationnelles générales lors de l'utilisation de s///, etc.

Tous types de pièges concernant les expressions rationnelles.

- s'\$lhs'\$rhs' n'effectue d'interpolation ni d'un côté ni de l'autre

s'\$lhs'\$rhs' n'effectue désormais d'interpolation ni d'un côté ni de l'autre. \$lhs était habituellement interpolé, mais pas \$rhs (et ne détecte toujours pas un '\$' littéral dans la chaîne)

```
$a=1;$b=2;
```

```
$string = '1 2 $a $b';
```

```
$string =~ s'$a'$b';
```

```
print $string,"\n";
```

```
# perl4 affiche : $b 2 $a $b
```

```
# perl5 affiche : 1 2 $a $b
```

- /m//g attache son état à la chaîne fouillée

m//g attache maintenant son état à la chaîne fouillée plutôt que l'expression rationnelle (une fois que la portée d'un bloc est quittée pour le sous-programme, l'état de la chaîne fouillée est perdu)

```
$_ = "ababab";
```

```
while(m/ab/g){
```

```
    &doit("blah");
```

```
}
```

```
sub doit{local($_) = shift; print "Got $_ "}
```

```
# perl4 affiche : Got blah Got blah Got blah Got blah
```

```
# perl5 affiche : boucle infinie de blah...
```

- m//o utilisé dans un sous-programme anonyme

Couramment, si vous utilisez le qualificateur m//o sur une expression rationnelle dans un sous-programme anonyme, toutes les fermetures générées dans ce sous-programme anonyme utiliseront l'expression rationnelle telle qu'elle a été compilée lors de sa toute première utilisation dans une telle fermeture. Par exemple, si vous dites

```
sub build_match {
```

```
    my($left,$right) = @_;
```

```

    return sub { $_[0] =~ /$left stuff $right/o; };
}
$good = build_match('foo', 'bar');
$bad = build_match('baz', 'blarch');
print $good->('foo stuff bar') ? "ok\n" : "not ok\n";
print $bad->('baz stuff blarch') ? "ok\n" : "not ok\n";
print $bad->('foo stuff bar') ? "not ok\n" : "ok\n";

```

Pour la plupart des distribution de Perl5, cela affichera :

```

ok
not ok
not ok

```

`build_match()` retournera toujours un sous-programme qui correspondra au contenu de `$left` et de `$right` tels qu'ils étaient la *première* fois que `build_match()` a été appelé, et pas tels qu'ils sont dans l'appel en cours.

- `$+` ne contient plus la chaîne reconnue

Si aucune parenthèse n'est utilisée dans une correspondance, Perl4 fixe `$+` à toute la correspondance, tout comme `&&`. Perl5 ne le fait pas.

```

"abcdef" =~ /b.*e/;
print "\$+ = \$+\n";
# perl4 affiche : bcde
# perl5 affiche :

```

- Une substitution retourne la chaîne vide si elle échoue

La substitution retourne désormais la chaîne vide si elle échoue

```

$string = "test";
$value = ($string =~ s/foo//);
print $value, "\n";
# perl4 affiche : 0
# perl5 affiche :

```

Voir aussi Pièges Numériques pour un autre exemple de cette nouvelle caractéristique.

- `s'lhs'rhs'` est une substitution normale

`s'lhs'rhs'` (en utilisant des accents graves) est maintenant une substitution normale, sans expansion des accents graves

```

$string = "";
$string =~ s'^hostname';
print $string, "\n";
# perl4 affiche : <the local hostname>
# perl5 affiche : hostname

```

- Analyse stricte des variables utilisées dans les expressions rationnelles

Analyse plus stricte des variables utilisées dans les expressions rationnelles

```

s/^([$grpc]*$grpc[$opt$plus$rep]?)/o;
# perl4: compile sans erreur
# perl5: compile avec l'erreur :
#   Scalar found where operator expected ..., near "$opt$plus"

```

un ajout à cet exemple, apparemment depuis le même script, est la véritable valeur de la chaîne après la substitution.

`[$opt]` est une classe de caractère en perl4 et un indice de tableau en perl5

```

$grpc = 'a';
$opt = 'r';
$_ = 'bar';
s/^([$grpc]*$grpc[$opt]?)/foo/;
print ;
# perl4 affiche : foo
# perl5 affiche : foobar

```

- `m?x?` ne correspond qu'une seule fois

Sous perl5, `m?x?` ne correspond qu'une fois, comme `?x?`. Sous perl4, cela correspondait plusieurs fois, comme `/x/` ou `m!x!`.

```

$test = "once";
sub match { $test =~ m?once?; }
&match();
if( &match() ) {
    # m?x? correspond plusieurs fois
    print "perl4\n";
} else {

```

```

    # m?x? correspond une seule fois
    print "perl5\n";
}

```

```

# perl4 affiche : perl4

```

```

# perl5 affiche : perl5

```

- L'échec d'une reconnaissance par expression rationnelle ne réinitialise pas les variables de reconnaissance. Contrairement à Ruby, l'échec d'une reconnaissance par expression rationnelle ne réinitialise pas les variables de reconnaissance (\$1, \$2, ..., \$', ...).

15.1.14 Pièges des sous-programmes, des signaux et des tris

Le groupe général de pièges entre Perl4 et Perl5 ayant un rapport avec les Signaux, les Tris, et leurs sous-programmes associés, ainsi que les pièges généraux liés aux sous-programmes. Cela inclut certains pièges spécifiques au système d'exploitation.

- Les barewords (mots simples) sont considérés comme des appels de sous-programmes. Les barewords (mots simples) qui étaient considérés comme des chaînes pour Perl sont maintenant considérés comme des appels de sous-programmes si un sous-programme ayant ce nom est déjà défini (le compilateur l'a déjà vu).

```

sub SeeYa { warn"Hasta la vista, baby!" }
$SIG{'TERM'} = SeeYa;
print "SIGTERM is now $SIG{'TERM'}\n";
# perl4 affiche : SIGTERM is now main'SeeYa
# perl5 affiche : SIGTERM is now main:1

```

Utilisez `-w` pour capturer celui-ci

- `reverse` n'est plus autorisé en tant que nom de sous-programme de tri. `reverse` n'est plus autorisé en tant que nom de sous-programme de tri.

```

sub reverse{ print "yup "; $a <=> $b }
print sort reverse (2,1,3);
# perl4 affiche : yup yup 123
# perl5 affiche : 123
# perl5 avertit (si -w) : Ambiguous call resolved as CORE::reverse()

```
- `warn()` ne vous laisse pas spécifier un handle de fichier. Bien qu'il ait `_toujours_` imprimé sur `STDERR`, `warn()` vous laissait spécifier un handle de fichier en perl4. Avec perl5, ce n'est plus le cas.

```

warn STDERR "Foo!";
# perl4 affiche : Foo!
# perl5 affiche : String found where operator expected

```

15.1.15 Pièges du système d'exploitation

- SysV réinitialise correctement un gestionnaire de signal (signal handler). Sous HPUX, et certains autres systèmes d'exploitation de la famille SysV, on devait réinitialiser tout handle de signal à l'intérieur de la fonction de gestion du signal, chaque fois qu'un signal était traité avec perl4. Avec perl5, la réinitialisation est désormais faite correctement. Tout code reposant sur un handler n'étant `_pas_` réinitialisé devra être réécrit.

Depuis la version 5.002, Perl utilise `sigaction()` sous SysV.

```

sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';
$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}
# perl4 (HPUX) affiche : Got INT...
# perl5 (HPUX) affiche : Got INT... Got INT...

```

- La fonction `seek()` sous SysV ajoute maintenant correctement
Sous les systèmes d'exploitation de la famille SysV, `seek()` sur un fichier ouvert en mode d'ajout >> fait désormais ce qu'il doit selon la page de manuel de `fopen()`. C'est-à-dire que lorsqu'un fichier est ouvert en mode d'ajout, il est impossible d'écraser les informations déjà présentes dans le fichier.

```
open(TEST, ">>seek.test");
$start = tell TEST ;
foreach(1 .. 9){
    print TEST "$_ ";
}
$end = tell TEST ;
seek(TEST, $start, 0);
print TEST "18 characters here";
# perl4 (solaris) seek.test dit : 18 characters here
# perl5 (solaris) seek.test dit : 1 2 3 4 5 6 7 8 9 18 characters here
```

15.1.16 Pièges de l'interpolation

Pièges entre Perl4 et Perl5 ayant un rapport avec la façon dont les choses sont interpolées dans certaines expressions, instructions, dans certains contextes, ou quoi que ce soit d'autre.

- Un `@` est désormais toujours interpolé en tant que tableau dans une chaîne entre guillemets
Un `@` est désormais toujours interpolé en tant que tableau dans une chaîne entre guillemets

```
print "To: someone@somewhere.com\n";
# perl4 affiche : To:someone@somewhere.com
# perl < 5.6.1, erreur : In string, @somewhere now must be written as \@somewhere
# perl >= 5.6.1, avertissement : Possible unintended interpolation of @somewhere in string
```

- Une chaîne entre guillemets ne peut plus se terminer par un `$` sans échappement
Les chaînes entre guillemets ne peuvent plus se terminer par un `$` non protégé par le caractère d'échappement.

```
$foo = "foo$";
print "foo is $foo\n";
# perl4 affiche : foo is foo$
# erreur de perl5 : Final $ should be \$ or $name
```

Note : perl5 NE génère PAS une erreur pour un `@` final.

- Une expression quelconque entre accolades dans une chaîne entre guillemets est évaluée
Perl évalue maintenant parfois des expressions quelconques placées entre accolades qui apparaissent entre des guillemets (habituellement lorsque l'accolade ouvrante est précédée par `$` ou `@`).

```
@www = "buz";
$foo = "foo";
$bar = "bar";
sub foo { return "bar" };
print "|@{w.w.w}|${main'foo}|";
# perl4 affiche : @{w.w.w}|foo|
# perl5 affiche : |buz|bar|
```

Notez que vous pouvez utiliser `use strict`; pour vous mettre à l'abri de tels pièges sous perl5.

- `$$x` tente de déréférencer `$x`
La construction "this is `$$x`" interpolait le pid à cet endroit, mais elle essaye maintenant de déréférencer `$x`. `$$` tout seul fonctionne toutefois toujours bien.

```
$s = "a reference";
$x = *s;
print "this is $$x\n";
# perl4 affiche : this is XXXx (XXX is the current pid)
# perl5 affiche : this is a reference
```

- La création d'une table de hachage au vol par un `eval` "EXPR" doit être protégée
La création de hachage à la volée par `eval` "EXPR" exige maintenant soit que les deux `$` soient protégés dans la spécification du nom du hachage, soit que les deux accolades soient protégées. Si les accolades sont protégées, le résultat sera compatible entre perl4 et perl5. C'est une pratique très commune qui devrait être modifiée de façon à utiliser si possible la forme bloc d'`eval{}`.

```
$hashname = "foobar";
$key = "baz";
$value = 1234;
eval "\$$hashname{'$key'} = q|$value|";
```

- ```

 (defined($foobar{'baz'})) ? (print "Yup") : (print "Nope");
 # perl4 affiche : Yup
 # perl5 affiche : Nope
En changeant
 eval "\$$hashname{'$key'} = q|$value|";
par
 eval "\$$hashname{'$key'} = q|$value|";
on obtient le résultat suivant :
 # perl4 affiche : Nope
 # perl5 affiche : Yup
ou en le changeant par
 eval "\$$hashname\{'$key'\} = q|$value|";
on obtient le résultat suivant :
 # perl4 affiche : Yup
 # perl5 affiche : Yup
 # et est compatible avec les deux versions

```
- Bogues des versions précédentes de perl
 

Certains programmes perl4 se reposaient inconsciemment sur les bugs de versions précédentes de perl.

```

perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'
perl4 affiche : This is not perl5
perl5 affiche : This is perl5

```
  - Interpolation de crochets (tableaux) et des accolades (tables de hachage)
 

Vous devez faire attention à l'interpolation des crochets (tableaux) et des accolades (tables de hachage).

```

print "$foo["
perl 4 affiche : [
perl 5 affiche : syntax error
print "$foo{"
perl 4 affiche : {
perl 5 affiche : syntax error

```

Perl 5 s'attend à trouver un indice (ou une clé) après le crochet ouvrant (ou l'accolade ouvrante) ainsi que le crochet fermant (l'accolade fermante) correspondant. Pour retrouver le comportement de Perl 4, vous devez protéger votre crochet (ou accolade) par le caractère d'échappement.

```

print "$foo\[";
print "$foo\{";

```
  - Interpolation de `\$$foo{bar}`

De façon similaire, prenez garde à `\$$foo{bar}`

```

$foo = "baz";
print "\$$foo{bar}\n";
perl4 affiche : $baz{bar}
perl5 affiche : $

```

Perl 5 cherche `$array{bar}` qui n'existe pas, mais perl 4 est content de simplement substituer `$foo` par "baz". Faites attention à cela, en particulier dans les `eval`'s.
  - Une chaîne qq() dans un eval ne trouve pas sa fin de chaîne
 

Une chaîne qq() passée à eval pose le problème suivant :

```

eval qq(
 foreach \$y (keys %$x) {
 \$count++;
 }
);
perl4 exécute cela sans problème
perl5 affiche : Can't find string terminator ")"

```

### 15.1.17 Pièges DBM

Pièges généraux de DBM.

- Perl5 doit être lié avec la même bibliothèque dbm/ndbm que la version par défaut de `dbmopen()`

Les bases de données existantes créées sous perl4 (ou tout autre outil dbm/ndbm) peuvent faire échouer le même script une fois exécuté sous perl5. L'exécutable de perl5 doit avoir été lié avec le même dbm/ndbm par défaut que `dbmopen()` pour qu'il fonctionne correctement sans lien via `tie` vers une implémentation dbm sous forme d'extension.

```

dbmopen (%dbm, "file", undef);

```

```

 print "ok\n";
 # perl4 affiche : ok
 # perl5 affiche : ok (IFF linked with -ldb or -lndbm)
- Un dépassement de la limite de taille d'une clé ou d'une valeur DBM arrête perl5 immédiatement
 Les bases de données existantes créées sous perl4 (ou tout autre outil dbm/ndbm) peuvent faire échouer le même script
 une fois exécuté sous perl5. L'erreur générée lorsque la limite de la taille clé/valeur est dépassée provoquera la sortie
 immédiate de perl5.
 dbmopen(DB, "testdb",0600) || die "couldn't open db! $!";
 $DB{'trap'} = "x" x 1024; # valeur trop grande pour la plupart
 # des dbm/ndbm

 print "YUP\n";
 # perl4 affiche :
 dbm store returned -1, errno 28, key "trap" at - line 3.
 YUP
 # perl5 affiche :
 dbm store returned -1, errno 28, key "trap" at - line 3.

```

### 15.1.18 Pièges non classés

Tout le reste.

- Piège require/do utilisant la valeur de retour

Si le fichier doit.pl contient :

```

sub foo {
 $rc = do "./do.pl";
 return 8;
}
print &foo, "\n";

```

Et le fichier do.pl contient l'unique ligne suivante :

```
return 3;
```

Exécuter doit.pl donne le résultat suivant :

```

perl 4 affiche : 3 (abandonne tôt le sous-programme)
perl 5 affiche : 8

```

Le comportement est le même si vous remplacez do par require.

- split sur une chaîne vide avec une LIMIT spécifiée

```

$string = '';
@list = split(/foo/, $string, 2)

```

Perl4 retourne une liste à un élément contenant la chaîne vide mais Perl5 retourne une liste vide.

Comme toujours, si certains de ces pièges sont déclarés un jour officiellement comme étant des bugs, ils seront corrigés et retirés.

## 15.2 TRADUCTION

### 15.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 15.2.2 Traducteur

Traduction initiale (5.6.0) : Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>. Mise à jour (5.8.8) : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 15.2.3 Relecture

Personne pour l'instant.

# Chapitre 16

## perldebtut

Tutoriel de débogage de Perl

### 16.1 DESCRIPTION

Une (très) légère introduction sur l'utilisation du débogueur de Perl, et un pointeur vers des sources d'information approfondies disponibles sur le sujet du débogage des programmes perl.

Il y a un nombre extraordinaire de personnes qui semblent ne rien connaître de l'utilisation de débogueur de Perl, bien qu'ils utilisent ce langage quotidiennement. Ceci est pour eux.

### 16.2 Utiliser strict (use strict)

Tout d'abord vous pouvez, sans utiliser le débogueur perl, mettre en oeuvre quelques idées pour vous simplifier la vie quand arrive le moment de déboguer des programmes perl. En guise de démonstration, voici un script simple, nommé "salut", présentant un problème :

```
#!/usr/bin/perl

$var1 = 'Salut le monde'; # toujours voulu faire cela :-)
$var2 = "$var1\n";

print $var2;
exit;
```

Bien que ce script se compile et s'exécute gaiement, il ne va probablement pas faire ce que l'on en attend, c'est à dire qu'il ne va pas du tout écrire "Salut le monde\n"; il va en revanche faire exactement ce que nous lui avons demandé de faire, les ordinateurs ayant légèrement tendance à agir de cette manière. Ce script va donc écrire un caractère de nouvelle ligne, et vous allez obtenir ce qui ressemble à une ligne blanche. Il semble que le script ne contiennent que deux variables, alors qu'il en contient en fait trois (à cause de la faute de frappe).

```
$var1 = 'Hello World';
$var1 = undef;
$var2 = "\n";
```

Afin de mettre en évidence ce type de problème, nous pouvons forcer la déclaration de chaque variable avant son utilisation en faisant appel au module « strict » par l'insertion après la première ligne du script de l'instruction 'use strict'. Maintenant, lors de l'exécution, perl se plaint de trois variables non déclarées, et nous obtenons quatre messages d'erreur car une variable est référencée deux fois :

```
Global symbol "$var1" requires explicit package name at ./t1 line 4.
Global symbol "$var2" requires explicit package name at ./t1 line 5.
Global symbol "$var1" requires explicit package name at ./t1 line 5.
Global symbol "$var2" requires explicit package name at ./t1 line 7.
Execution of ./hello aborted due to compilation errors.
```



Merveilleux ! Et pour supprimer ces erreurs, nous déclarons explicitement toutes les variables. Maintenant notre script ressemble à ceci :

```
#!/usr/bin/perl
use strict;

my $var1 = 'Salut le monde';
my $var1 = undef;
my $var2 = "$var1\n";

print $var2;
exit;
```

Nous procédons alors à une vérification de la syntaxe (toujours une bonne idée) avant d'exécuter à nouveau le script :

```
> perl -c hello
hello syntax OK
```

Et maintenant quand nous exécutons ce script, nous obtenons toujours "\n", mais au moins nous savons pourquoi. La compilation du script a révélé la variable '\$var1' (avec la lettre 'l') et le simple changement de \$varl par \$var1 résoud le problème.

### 16.3 Observer les variables et -w et v

D'accord, mais comment faire lorsque vous souhaitez réellement voir vos données, le contenu d'une variable dynamique, juste avant son utilisation ?

```
#!/usr/bin/perl
use strict;

my $key = 'welcome';
my %data = (
 'this' => qw(that),
 'tom' => qw(and jerry),
 'welcome' => q>Hello World),
 'zip' => q(welcome),
);
my @data = keys %data;

print "$data{$key}\n";
exit;
```

Tout semble correct. Après avoir été testé avec la vérification de syntaxe (**perl -c nom\_du\_script**) nous exécutons ce script et tout ce que nous obtenons est à nouveau une ligne blanche ! Hmmmmm. Dans ce cas, une approche courante du débogage serait de parsemer délibérément quelques instructions print, pour ajouter une première vérification juste avant d'écrire nos données, et une seconde juste après :

```
print "All OK\n" if grep($key, keys %data);
print "$data{$key}\n";
print "done: '$data{$key}'\n";
```

Après une nouvelle tentative :

```
> perl data
All OK

done: ''
```

Après s'être usé les yeux sur ce code pendant un bon moment, nous allons boire une tasse de café et nous tentons une nouvelle approche. C'est-à-dire que nous appelons la cavalerie en invoquant perl avec l'option '-d' sur la ligne de commande.

```
> perl -d data
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(./data:4): my $key = 'welcome';
```

Donc, ce que nous venons de faire ici est de lancer notre script avec le débogueur intégré de perl. Celui-ci s'est arrêté à la première ligne de code exécutable et attend notre action.

Avant de poursuivre plus avant, vous allez souhaiter connaître comment quitter le débogueur : tapez simplement la lettre 'q', pas les mots 'quit', ni 'exit'.

```
DB<1> q
>
```

C'est cela, vous êtes à nouveau au paddock.

## 16.4 Aide (help)

Démarrer à nouveau le débogueur et nous allons examiner le menu d'aide. Il y a plusieurs méthodes pour afficher l'aide : taper simplement 'h' affichera une longue liste déroulante, 'lh' (pipe-h) redirigera l'aide vers l'afficheur page par page ('more' ou 'less' probablement), et finalement, 'h h' (h-espace-h) vous donnera un écran très utile résumant les commandes :

```
DB<1> h
List/search source lines: Control script execution:
l [ln|sub] List source code T Stack trace
- or . List previous/current line s [expr] Single step [in expr]
v [line] View around line n [expr] Next, steps over subs
f filename View source in file <CR/Enter> Repeat last n or s
/pattern/ ?patt? Search forw/backw r Return from subroutine
M Show module versions c [ln|sub] Continue until position
Debugger controls: L List break/watch/actions
o [...] Set debugger options t [expr] Toggle trace [trace expr]
<[<]|{[{}]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cmd] Set breakpoint
! [N|pat] Redo a previous command B ln|* Delete a/all breakpoints
H [-num] Display last num commands a [ln] cmd Do cmd before line
= [a val] Define/list an alias A ln|* Delete a/all actions
h [db_cmd] Get help on command w expr Add a watch expression
h h Complete help page W expr|* Delete a/all watch exprs
|[[]db_cmd Send output to pager ![!] syscmd Run cmd in a subprocess
q or ^D Quit R Attempt a restart

Data Examination: expr Execute perl code, also see: s,n,t expr
x|m expr Evals expr in list context, dumps the result or lists methods.
p expr Print expression (uses script's current package).
S [![pat] List subroutine names [not] matching pattern
V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
X [Vars] Same as "V current_package [Vars]". i class inheritance tree.
y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.
e Display thread id E Display all thread ids.

For more help, type h cmd_letter, or run man perldebug for all docs.
```

Plus d'options que vous ne le pensiez ! Ce n'est pas si terrible qu'il y paraît, et il est très utile et en plus amusant, d'en connaître plus à propos de chacune de ces commandes.

Il y a quelques options utiles à connaître immédiatement. Vous pensez probablement que vous n'utilisez aucune bibliothèque en ce moment, mais 'M' va vous montrer les modules actuellement actifs, ceux appelés aussi bien par le débogueur que par le script. De même, 'm' vous affichera les méthodes et 'S' tous les sous-programmes (par motif) comme ci-dessous. 'V' et 'X' montrent les variables du programme rangées par package et peuvent être restreintes par l'utilisation de motifs.

```
DB<2>S str
dumpvar::stringify
strict::bits
strict::import
strict::unimport
```

L'utilisation de 'X' et de ses cousins demande de ne pas utiliser l'identificateur de type (\$@%), seulement le 'nom' :

```
DM<3>X ~err
FileHandle(stderr) => fileno(2)
```

Rappelez-vous que nous sommes dans notre minuscule programme avec un problème, nous devrions jeter un oeil pour voir où nous sommes et à quoi nos données ressemblent. Avant tout, regardons un peu de code autour de notre position actuelle (la première ligne de code dans notre cas), grâce à la lettre 'v' :

```
DB<4> v
1 #!/usr/bin/perl
2: use strict;
3
4==> my $key = 'welcome';
5: my %data = (
6 'this' => qw(that),
7 'tom' => qw(and jerry),
8 'welcome' => q>Hello World),
9 'zip' => q>welcome),
10);
```

À la ligne 4 se présente un pointeur utile, qui vous indique où vous vous situez actuellement. Pour voir plus avant dans le programme, tapons encore 'v'.

```
DB<4> w
8 'welcome' => q>Hello World),
9 'zip' => q>welcome),
10);
11: my @data = keys %data;
12: print "All OK\n" if grep($key, keys %data);
13: print "$data{$key}\n";
14: print "done: '$data{$key}'\n";
15: exit;
```

Et si vous voulez lister à nouveau la ligne 5, tapez 'l 5', (remarquez l'espace) :

```
DB<4> l 5
5: my %data = (
```

Dans le cas de ce script, il n'y pas grand chose à voir, mais il peut y avoir des pages entières à explorer et 'I' peut être très utile. Pour réinitialiser votre vue sur la ligne qui va s'exécuter, tapez un simple point '.' :

```
DB<5> .
main:(./data_a:4): my $key = 'welcome';
```

La ligne affichée est la prochaine qui sera exécutée, sans que cette exécution ne se soit produite pour l'instant. Donc bien que nous puissions imprimer une variable grâce à la commande 'p', tout ce que nous allons obtenir est une valeur vide (non définie). Ce que nous devons faire est exécuter la prochaine instruction exécutable en tapant 's' :

```
DB<6> s
main::(/data_a:5): my %data = (
main::(/data_a:6): 'this' => qw(that),
main::(/data_a:7): 'tom' => qw(and jerry),
main::(/data_a:8): 'welcome' => q>Hello World),
main::(/data_a:9): 'zip' => q>welcome),
main::(/data_a:10):);
```

Maintenant, nous pouvons observer cette première variable (\$clef)

```
DB<7> p $key
welcome
```

La ligne 13 est celle où l'action a lieu, donc continuons notre progression grâce à la lettre 'c', qui insert un point d'arrêt 'à usage unique' à la ligne ou au sous-programme indiqué.

```
DB<8> c 13
All OK
main::(/data_a:13): print "$data{$key}\n";
```

Nous avons dépassé notre vérification (où 'All OK' était imprimé) et nous nous sommes arrêtés juste avant le gros du travail. Nous pourrions essayer de lister quelques variables pour voir ce qu'il se passe.

```
DB<9> p $data{$key}
```

Pas grand chose ici, observons notre hachage :

```
DB<10> p %data
Hello Worldziptomandwelcomejerrywelcomethisthat

DB<11> p keys %data
Hello Worldtomwelcomejerrythis
```

Bon, ceci n'est pas très facile à lire, et dans l'écran d'aide (**h h**), la commande 'x' semble prometteuse :

```
DB<12> x %data
0 'Hello World'
1 'zip'
2 'tom'
3 'and'
4 'welcome'
5 undef
6 'jerry'
7 'welcome'
8 'this'
9 'that'
```

Ceci n'aide pas beaucoup, deux 'bienvenue', mais sans que l'on sache s'ils correspondent à des clefs ou à des valeurs. Il s'agit juste de l'affichage d'une image d'un tableau, dans ce cas sans grande utilité. L'astuce est ici d'utiliser une référence à la structure de données :

```
DB<13> x \%data
0 HASH(0x8194bc4)
 'Hello World' => 'zip'
 'jerry' => 'welcome'
 'this' => 'that'
 'tom' => 'and'
 'welcome' => undef
```

La référence est réellement détaillée, et nous mettons finalement en évidence notre problème. Notre citation est parfaitement valide, mais erronée pour notre propos, avec 'and jerry' traité comme deux mots séparés plutôt que comme une phrase, donc entraînant un mauvais alignement des paires du hachage.

Le commutateur '-w' nous aurait indiqué ce problème si nous l'avions utilisé au départ, et nous aurait évité pas mal de soucis :

```
> perl -w data
Odd number of elements in hash assignment at ./data line 5.
```

Nous réparons notre citation : 'tom' =>q(et jerry), et exécutons à nouveau notre programme. Nous obtenons alors le résultat attendu :

```
> perl -w data
Bonjour le monde
```

Pendant que nous y sommes, regardons de plus près la commande 'x'. Elle est réellement utile et vous affichera merveilleusement le contenu de références imbriquées, d'objets complets, de fragments d'objets, de tout ce que vous voudrez bien lui appliquer :

Construisons rapidement un objet, et e-x-plorons le. D'abord, démarrons le débogueur : il demande une entrée à partir de STDIN, donnons-lui donc quelque chose qui n'engage à rien, un zéro :

```
> perl -de 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(-e:1): 0
```

Maintenant construisez au vol un objet sur quelques lignes (notez l'antislash) :

```
DB<1> $obj = bless({'unique_id'=>'123', 'attr'=> \
cont: {'col' => 'black', 'things' => [qw(this that etc)]}, 'MY_class')
```

Et jetez un oeil dessus :

```
DB<2> x $obj
0 MY_class=HASH(0x828ad98)
 'attr' => HASH(0x828ad68)
 'col' => 'black'
 'things' => ARRAY(0x828abb8)
 0 'this'
 1 'that'
 2 'etc'
 'unique_id' => 123
DB<3>
```

Utile, n'est-ce pas ? Vous pouvez utiliser eval sur n'importe quoi, et expérimenter avec des fragments de programme ou d'expressions régulières jusqu'à ce que les vaches rentrent à l'étable :

```
DB<3> @data = qw(this that the other atheism leather theory scythe)

DB<4> p 'saw -> ' . ($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 6
```

Si vous voulez voir l'historique des commandes, tapez un 'H' :

```
DB<5> H
4: p 'saw -> ' .($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
3: @data = qw(this that the other atheism leather theory scythe)
2: x $obj
1: $obj = bless({'unique_id'=>'123', 'attr'=>
{'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
DB<5>
```

Et si vous voulez répéter une quelques commande précédente, utilisez le point d'exclamation '!' :

```
DB<5> !4
p 'saw -> ' .($cnt += map { print "$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 12
```

Pour plus d'informations sur les références, voyez les pages de manuel perlref et perlrefut.

## 16.5 Pas à pas dans le code

Voici un simple programme qui convertit les degrés Fahrenheit en Celsius (et vice-versa). Ce programme présente aussi un problème.

```
#!/usr/bin/perl -w
use strict;

my $arg = $ARGV[0] || '-c20';

if ($arg =~ /^^(c|f)((\-|\+)*\d+(\.\d+)*$)/ {
 my ($deg, $num) = ($1, $2);
 my ($in, $out) = ($num, $num);
 if ($deg eq 'c') {
 $deg = 'f';
 $out = &c2f($num);
 } else {
 $deg = 'c';
 $out = &f2c($num);
 }
 $out = sprintf('%0.2f', $out);
 $out =~ s/^(c|f)((\-|\+)*\d+)\.0+$/\d/;
 print "$out $deg\n";
} else {
 print "Usage: $0 -[c|f] num\n";
}
exit;

sub f2c {
 my $f = shift;
 my $c = 5 * $f - 32 / 9;
 return $c;
}
```

```

sub c2f {
 my $c = shift;
 my $f = 9 * $c / 5 + 32;
 return $f;
}

```

Pour une raison inconnue, la conversion des Fahrenheit en Celsius ne donne pas le résultat escompté. Voici ce que ce programme effectue :

```

> temp -c0.72
33.30 f

> temp -f33.3
162.94 c

```

Pas très cohérent ! Nous allons placer manuellement un point d'arrêt dans le programme et exécuter celui-ci dans le débogueur pour voir ce qui se passe. Un point d'arrêt est un drapeau. Le débogueur va exécuter le programme sans interruption jusqu'à ce drapeau. Quand il l'atteint, il arrête l'exécution et présente une invite pour la suite de l'interaction. En utilisation normale, ces commandes de débogage sont totalement ignorées, et elle peuvent être conservées en toute sécurité dans le programme final, bien qu'elles en augmentent quelque peu le désordre.

```

my ($in, $out) = ($num, $num);
$DB::single=2; # insert at line 9!
if ($deg eq 'c')
 ...

```

```

> perl -d temp -f33.3
Default die handler restored.

```

```

Loading DB routines from perl5db.pl version 1.07
Editor support available.

```

```

Enter h or 'h h' for help, or 'man perldebug' for more help.

```

```

main::(temp:4): my $arg = $ARGV[0] || '-c20';

```

Nous allons simplement descendre jusqu'à notre point d'arrêt prédéfini grâce à la commande 'c' :

```

DB<1> c
main::(temp:10): if ($deg eq 'c') {

```

suivie par une commande 'v' pour voir où nous sommes :

```

DB<1> v
7: my ($deg, $num) = ($1, $2);
8: my ($in, $out) = ($num, $num);
9: $DB::single=2;
10==> if ($deg eq 'c') {
11: $deg = 'f';
12: $out = &c2f($num);
13: } else {
14: $deg = 'c';
15: $out = &f2c($num);
16: }

```

et par une commande 'p' pour montrer le contenu des variables que nous utilisons :

```

DB<1> p $deg, $num
f33.3

```

Nous pouvons placer un autre point d'arrêt à n'importe quelle ligne commençant par un numéro suivi par deux points. Nous utiliserons la ligne 17, cette ligne étant la première après la sortie du sous-programme f2c. Nous prévoyons de faire une pause à cet endroit plus tard.

```
DB<2> b 17
```

Il n'y a pas d'indications en retour, mais vous pouvez voir les points d'arrêt placés en utilisant la commande 'L'.

```
DB<3> L
temp:
 17: print "$out $deg\n";
 break if (1)
```

Il est à noter que les points d'arrêts peuvent être supprimés en utilisant les commandes 'd' ou 'D'.

Nous allons alors continuer plus avant dans notre sous-programme, cette fois en utilisant, à la place du numéro de ligne, le nom du sous-programme suivi par la commande 'v', maintenant familière,

```
DB<3> c f2c
main::f2c(temp:30): my $f = shift;

DB<4> v
24: exit;
25
26 sub f2c {
27==> my $f = shift;
28: my $c = 5 * $f - 32 / 9;
29: return $c;
30 }
31
32 sub c2f {
33: my $c = shift;
```

Notez que si il existait un appel à un sous-programme entre notre position et la ligne 29, et si nous voulions traverser ce sous-programme pas-à-pas, nous pourrions utiliser la commande 's'. Pour le franchir, il faudrait utiliser 'n' qui exécuterait alors le sous-programme sans l'inspecter. Dans notre cas, nous continuons simplement jusqu'à la ligne 29.

Regardons la valeur retournée :

```
DB<5> p $c
162.9444444444444
```

Ce n'est pas du tout la réponse exacte, mais l'opération semble correcte. Je me demande si le problème ne serait pas lié aux priorités des opérateurs ? Essayons notre opération avec quelques autres combinaisons de parenthèses.

```
DB<6> p (5 * $f - 32 / 9)
162.9444444444444
```

```
DB<7> p 5 * $f - (32 / 9)
162.9444444444444
```

```
DB<8> p (5 * $f) - 32 / 9
162.9444444444444
```

```
DB<9> p 5 * ($f - 32) / 9
0.722222222222221
```

:-) Ce dernier essai ressemble plus à ce que l'on souhaite ! Bien, maintenant nous pouvons définir la variable à renvoyer, et nous allons sortir de notre sous-programme grâce à la commande 'r' :

```
DB<10> $c = 5 * ($f - 32) / 9
```



```
DB<11> r
scalar context return from main::f2c: 0.72222222222221
```

Semble correct, continuons donc jusqu'à la fin du script :

```
DB<12> c
0.72 c
Debugged program terminated. Use q to quit or R to restart,
use 0 inhibit_exit to avoid stopping after program termination,
h q, h R or h 0 to get additional info.
```

Une correction rapide de la ligne erronée (insertion des parenthèses manquantes) dans le programme et nous avons terminé.

## 16.6 Emplacement réservé pour a, w, t, T

Action, observation des variables, suivi des piles etc...: sur la liste des choses à faire.

```
a
w
t
T
```

## 16.7 Expressions rationnelles (ou régulières)

Avez-vous jamais voulu savoir à quoi ressemble une expression régulière ? Vous aurez besoin pour cela de compiler Perl avec le drapeau DEBUGGING :

```
> perl -Dr -e '/^pe(a)*rl$/i'
Compiling REx '^pe(a)*rl$'
size 17 first at 2
rarest char
at 0
 1: BOL(2)
 2: EXACTF (4)
 4: CURLYN[1] {0,32767}(14)
 6: NOTHING(8)
 8: EXACTF (0)
 12: WHILEM(0)
 13: NOTHING(14)
 14: EXACTF (16)
 16: EOL(17)
 17: END(0)
floating '$' at 4..2147483647 (checking floating) stclass 'EXACTF '
anchored(BOL) minlen 4
Omitting '$ $& $' support.

EXECUTING...

Freeing REx: '^pe(a)*rl$'
```

Voulez-vous vraiment savoir ?:-) Pour plus de détails sanglants sur la mise en oeuvre des expressions régulières, regardez les pages de manuel perlre, perlretut, et pour décoder les étiquettes mystérieuses (voir ci-dessus BOL, CURLYN etc..), se référer au manuel perldebguts.

## 16.8 Conseils sur les sorties

Pour obtenir toutes les sorties de votre journal d'erreur, et ne manquer aucun messages via les tampons du système d'exploitation, insérez au début de votre script une ligne de ce type :

```
$|=1;
```

Pour voir l'extrémité d'un fichier journal en croissance dynamique, tapez à partir de la ligne de commande :

```
tail -f $error_log
```

Envelopper tous les appels à 'die' dans une routine peut être utilisé pour voir comment et à partir de quoi ils sont appelés. La page de manuel perlvar présente plus d'informations :

```
BEGIN { $SIG{__DIE__} = sub { require Carp; Carp::confess(@_) } }
```

Diverses techniques utilisées pour la redirection des descripteurs STDOUT et STDERR sont expliquées dans les pages de manuel perlperentut et perlfaq8.

## 16.9 CGI

Juste un petit truc pour tous les programmeurs CGI qui ne trouvent comment poursuivre après l'invite 'waiting for input' quand ils exécutent leur script à partir de la ligne de commande. Essayez quelque chose comme :

```
> perl -d my_cgi.pl -nodebug
```

Bien sûr les pages de manuel CGI et perlfaq9 vous en dirons plus.

## 16.10 Interfaces graphiques (GUI)

L'interface de la ligne de commande est étroitement intégrée avec une extension à emacs et il existe aussi une interface vers vi.

Vous n'avez pas à faire tout ceci à partir de la ligne de commande, il existe quelques possibilités d'utilisation d'interfaces graphiques. Ce qui est agréable alors est de pouvoir agiter la souris au dessus d'une variable et de voir apparaître dans une fenêtre appropriée ou dans un ballon d'aide son contenu. Plus besoin de se fatiguer à taper 'x \$ **nomdevariable**' :-)

En particulier, vous pouvez partir à la chasse de :

**ptkdb** Enveloppe pour le débogueur intégré basée sur perlTK

**ddd** data display debugger (débogueur exposeur de données)

**PerlDevKit** et **PerlBuilder** sont des outils spécifiques à Windows NT.

NB. Plus d'informations sur ces outils ainsi que sur d'autres équivalents seraient appréciées.

## 16.11 Résumé

Nous avons vu comment encourager de bonnes pratiques de programmation grâce à l'utilisation de **use strict** et de **-w**. Nous pouvons exécuter le débogueur de perl (**perl -d nomdescript**) pour inspecter nos données grâce aux commandes **p** et **x**. Vous pouvez parcourir votre code, placer des points d'arrêt avec la commande **b**, parcourir ce programme pas à pas avec **s** ou **n**, continuer avec **c** et revenir d'un sous-programme avec **r**. Plutôt intuitif quand on y regarde de près.

Il y a bien sûr beaucoup plus à découvrir à propos du débogueur, ce tutoriel n'ayant fait qu'explorer la surface. La meilleure façon d'en apprendre plus est d'utiliser perldoc pour découvrir beaucoup d'autres d'aspects du langage, de lire l'aide en ligne (la page de manuel perlbug devrait être votre prochaine lecture) et bien sûr d'expérimenter.

## 16.12 VOIR AUSSI

Les pages de manuel perldebug, perldebguts, perldiag, perlrun, *dprofpp*.

## 16.13 AUTEUR

Richard Foley <richard@rfi.net> Copyright(c) 2000

## 16.14 CONTRIBUTIONS

Diverses personnes ont aidées par leurs suggestions et leurs contributions, en particulier :

Ronald J Kimball <jk@linguist.dartmouth.edu>

Hugo van der Sanden <hv@crypt0.demon.co.uk>

Peter Scott <Peter@PSDT.com>

## 16.15 TRADUCTION

### 16.15.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 16.15.2 Traducteur

Traduction initiale : Landry Le Chevanton <landry.lechevanton@wanadoo.fr>. Mise à jour : Paul Gaborit <paul.gaborit@enstimac.fr>.

### 16.15.3 Relecture

Gérard Delafond

# Chapitre 17

## perlfaq

Foire aux questions sur Perl

### 17.1 DESCRIPTION

La FAQ Perl comprend plusieurs documents qui répondent aux questions les plus courantes concernant Perl et la programmation en Perl. Elle est organisée en neuf thèmes majeurs présentés ici.

#### 17.1.1 Où se procurer ce document ?

La FAQ Perl est fourni dans la distribution standard de Perl. Donc, si vous avez Perl, vous devez avoir la FAQ Perl. Vous devez aussi avoir l'outil `perldoc` qui vous permet de lire toute la documentation :

```
$ perldoc perlfaq
```

Au-delà de votre système local, vous pouvez trouver cette FAQ Perl sur le web. Par exemple sur <http://perldoc.perl.org/> (ou <http://perl.enstimac.fr/> pour la présente traduction française).

#### 17.1.2 Comment contribuer à ce document.

Les corrections, les ajouts ou les suggestions pour la version originale doivent être adressés à

`<perlfaq-workers AT perl DOT com>`. Les volontaires utilisent cette adresse pour coordonner leurs efforts et suivre le développement de la FAQ Perl. Ils sauront apprécier vos contributions mais ils n'ont pas le temps de vous aider, n'utilisez donc pas cette adresse pour poser des questions sur la FAQ Perl.

Les corrections et les ajouts à la version française doivent être adressés aux traducteurs ou à `<paul.gaborit AT enstimac DOT fr>`. Si vous voulez participer à l'effort de traduction et de relecture, vous pouvez passer par le même canal.

#### 17.1.3 Que va-t-il se passer si vous envoyez votre problème de Perl aux auteurs ?

Ne faites pas cela !

Si vous avez une question dont la réponse ne semble pas être dans la FAQ, vous feriez mieux de lire le chapitre 2 de cette FAQ pour savoir où poser votre question.

### 17.2 CREDITS

Tom Christiansen a écrit la FAQ Perl d'origine qui a ensuite été étendue avec l'aide de Nat Torkington. Les `perlfaq-workers` maintiennent les versions actuelles et font des mises à jour. De nombreuses personnes ont contribué aux questions, aux corrections et aux commentaires et la FAQ indique ces contributions à l'endroit approprié.

## 17.3 Author and Copyright Information

Tom Christiansen a écrit la FAQ Perl d'origine brian d foy <bdfoy@cpan.org> a écrit cette version. Voir les documents eux-mêmes pour les informations précises des droits.

Ce document est disponible sous les mêmes conditions que Perl lui-même. Les exemples de codes de tous les documents de la FAQ Perl sont dans le domaine public. Utilisez-les comme vous le souhaitez (à vos risques et périls).

## 17.4 Table des matières

*perlfqa* - Ce document.

*perlfqa1* - Questions générales sur Perl.

*perlfqa2* - Trouver et apprendre Perl.

*perlfqa3* - Outils de programmation.

*perlfqa4* - Manipulation de données.

*perlfqa5* - Fichiers et formats.

*perlfqa6* - Expressions rationnelles.

*perlfqa7* - Questions générales sur le langage Perl.

*perlfqa8* - Interaction avec le système.

*perlfqa9* - Réseau.

## 17.5 Toutes les questions

*perlfqa1*: Questions générales sur Perl.

Question et informations très générales sur Perl.

- Qu'est ce que Perl ?
- Qui supporte Perl ? Qui le développe ? Pourquoi est-il gratuit ?
- Quelle version de Perl dois-je utiliser ?
- Qu'est-ce que veut dire perl4, perl5 ou perl6 ?
- Qu'est-ce que Ponie ?
- Qu'est-ce que perl6 ?
- Est-ce que Perl est stable ?
- Est-il difficile d'apprendre Perl ?
- Est-ce que Perl tient la comparaison avec d'autres langages comme Java, Python, REXX, Scheme ou Tcl ?
- Que puis-je faire avec Perl ?
- Quand ne devrais-je pas programmer en Perl ?
- Quelle est la différence entre "perl" et "Perl" ?
- Parle-t-on de programme Perl ou de script Perl ?
- Qu'est ce qu'un JAPH ?
- Où peut on trouver une liste des mots d'esprit de Larry
- Comment convaincre mon administrateur système/chef de projet/employés d'utiliser Perl/Perl5 plutôt qu'un autre langage ?

*perlfqa2*: Trouver et apprendre Perl.

Où trouver les sources et la documentation de Perl ainsi que de l'aide et autres choses similaires.

- Quelles machines supportent perl ? Où puis-je trouver Perl ?
- Comment trouver une version binaire de perl ?
- Je n'ai pas de compilateur C sur mon système. Comment puis-je compiler mon propre interpréteur Perl ?
- J'ai copié le binaire perl d'une machine sur une autre mais les scripts ne fonctionnent pas.
- J'ai récupéré les sources et j'essaie de les compiler mais gdbm/dynamic loading/malloc/linking/... échoue. Comment faire pour que ça marche ?
- Quels modules et extensions existent pour Perl ? Qu'est-ce que CPAN ? Que signifie CPAN/src/... ?
- Existe-t-il une version de Perl certifiée ISO ou ANSI ?
- Où puis-je trouver des informations sur Perl ?
- Quels sont les groupes de discussion concernant Perl sur Usenet ? Où puis-je poser mes questions ?
- Où puis-je poster mon code source ?
- Les livres sur Perl

- Quels sont les revues ou magazines parlant de Perl ?
- Quelles sont les listes de diffusion concernant Perl ?
- Où trouver les archives de comp.lang.perl.misc ?
- Où puis-je acheter une version commerciale de Perl ?
- Où dois-je envoyer mes rapports de bugs ?
- Qu'est-ce que perl.com ? Les Perl Mongers ? pm.org ? perl.org ? cpan.org ?

### **perlfq3: Outils de programmation.**

Outils pour le programmeur et support technique.

- Comment fais-je pour... ?
- Comment utiliser Perl de façon interactive ?
- Existe-t-il un shell Perl ?
- Comment puis-je connaître les modules installés sur mon système ?
- Comment déboguer mes programmes Perl ?
- Comment mesurer les performances de mes programmes Perl ?
- Comment faire une liste croisée des appels de mon programme Perl ?
- Existe-t-il un outil de mise en page de code Perl ?
- Existe-t-il un ctags pour Perl ?
- Existe-t-il un environnement de développement intégré (IDE) ou un éditeur Perl sous Windows ?
- Où puis-je trouver des macros pour Perl sous vi ?
- Où puis-je trouver le mode perl pour emacs ?
- Comment utiliser des 'curses' avec Perl ?
- Comment puis-je utiliser X ou Tk avec Perl ?
- Comment rendre mes programmes Perl plus rapides ?
- Comment faire pour que mes programmes Perl occupent moins de mémoire ?
- Est-ce sûr de retourner un pointeur sur une donnée locale ?
- Comment puis-je libérer un tableau ou une table de hachage pour réduire mon programme ?
- Comment rendre mes scripts CGI plus efficaces ?
- Comment dissimuler le code source de mon programme Perl ?
- Comment compiler mon programme Perl en code binaire ou C ?
- Comment compiler Perl pour en faire du Java ?
- Comment faire fonctionner #!perl sur [MS-DOS,NT,...] ?
- Puis-je écrire des programmes Perl pratiques sur la ligne de commandes ?
- Pourquoi les commandes Perl à une ligne ne fonctionnent-elles pas sur mon DOS/Mac/VMS ?
- Où puis-je en apprendre plus sur la programmation CGI et Web en Perl ?
- Où puis-je apprendre la programmation orientée objet en Perl ?
- Où puis-je en apprendre plus sur l'utilisation liée de Perl et de C ?
- J'ai lu perlembed, perlguys, etc., mais je ne peux inclure du perl dans mon programme C, qu'est ce qui ne va pas ?
- Quand j'ai tenté d'exécuter mes scripts, j'ai eu ce message. Qu'est ce que cela signifie ?
- Qu'est-ce que MakeMaker ?

### **perlfq4: Manipulation de données.**

Les questions liées à la manipulation des nombres, des dates, des chaînes de caractères, des tableaux, des tables de hachage, ainsi qu'à divers problèmes relatifs aux données.

- Pourquoi est-ce que j'obtiens des longs nombres décimaux (ex. 19.9499999999999) à la place du nombre que j'attends (ex. 19.95) ?
- Pourquoi int() ne fonctionne pas bien ?
- Pourquoi mon nombre octal n'est-il pas interprété correctement ?
- Perl a-t-il une fonction round() ? Et ceil() (majoration) et floor() (minoration) ? Et des fonctions trigonométriques ?
- Comment faire des conversions numériques entre différentes bases, entre différentes représentations ?
- Pourquoi & ne fonctionne-t-il pas comme je le veux ?
- Comment multiplier des matrices ?
- Comment effectuer une opération sur une série d'entiers ?
- Comment produire des chiffres romains ?
- Pourquoi mes nombres aléatoires ne sont-ils pas aléatoires ?
- Comment obtenir un nombre aléatoire entre X et Y ?
- Comment trouver le jour ou la semaine de l'année ?
- Comment trouver le siècle ou le millénaire actuel ?
- Comment comparer deux dates ou en calculer la différence ?
- Comment convertir une chaîne de caractères en secondes depuis l'origine des temps ?
- Comment trouver le jour du calendrier Julien ?

- Comment trouver la date d'hier ?
- Perl a-t-il un problème avec l'an 2000 ? Perl est-il compatible an 2000 ?
- Comment m'assurer de la validité d'une entrée ?
- Comment supprimer les caractères d'échappement d'une chaîne de caractères ?
- Comment enlever des paires de caractères successifs ?
- Comment effectuer des appels de fonction dans une chaîne ?
- Comment repérer des éléments appariés ou imbriqués ?
- Comment inverser une chaîne de caractères ?
- Comment développer les tabulations dans une chaîne de caractères ?
- Comment remettre en forme un paragraphe ?
- Comment accéder à ou modifier N caractères d'une chaîne de caractères ?
- Comment changer la nième occurrence de quelque chose ?
- Comment compter le nombre d'occurrences d'une sous-chaîne dans une chaîne de caractères ?
- Comment mettre en majuscule toutes les premières lettre des mots d'une ligne ?
- Comment découper une chaîne séparée par un [caractère] sauf à l'intérieur d'un [caractère] ?
- Comment supprimer des espaces blancs au début/à la fin d'une chaîne ?
- Comment cadrer une chaîne avec des blancs ou un nombre avec des zéros ?
- Comment extraire une sélection de colonnes d'une chaîne de caractères ?
- Comment calculer la valeur soundex d'une chaîne ?
- Comment interpoler des variables dans des chaînes de texte ?
- En quoi est-ce un problème de toujours placer "\$vars" entre guillemets ?
- Pourquoi est-ce que mes documents <<HERE ne marchent pas ?
- Quelle est la différence entre une liste et un tableau ?
- Quelle est la différence entre \$array[1] et @array[1] ?
- Comment supprimer les doublons d'une liste ou d'un tableau ?
- Comment savoir si une liste ou un tableau inclut un certain élément ?
- Comment calculer la différence entre deux tableaux ? Comment calculer l'intersection entre deux tableaux ?
- Comment tester si deux tableaux ou hachages sont égaux ?
- Comment trouver le premier élément d'un tableau vérifiant une condition donnée ?
- Comment gérer des listes chaînées ?
- Comment gérer des listes circulaires ?
- Comment mélanger le contenu d'un tableau ?
- Comment traiter/modifier chaque élément d'un tableau ?
- Comment sélectionner aléatoirement un élément d'un tableau ?
- Comment générer toutes les permutations des N éléments d'une liste ?
- Comment trier un tableau par (n'importe quoi) ?
- Comment manipuler des tableaux de bits ?
- Pourquoi defined() retourne vrai sur des tableaux et hachages vides ?
- Comment traiter une table entière ?
- Que se passe-t-il si j'ajoute ou j'enlève des clefs d'un hachage pendant que j'itère dessus ?
- Comment rechercher un élément d'un hachage par sa valeur ?
- Comment savoir combien d'entrées sont dans un hachage ?
- Comment trier une table de hachage (par valeur ou par clef) ?
- Comment conserver mes tables de hachage dans l'ordre ?
- Quelle est la différence entre "delete" et "undef" pour des tables de hachage ?
- Pourquoi mes tables de hachage liées (par tie()) ne font pas la distinction entre exists et defined ?
- Comment réinitialiser une opération each() non terminée ?
- Comment obtenir l'unicité des clefs de deux hachages ?
- Comment enregistrer un tableau multidimensionnel dans un fichier DBM ?
- Comment faire en sorte que mon hachage conserve l'ordre des éléments que j'y mets ?
- Pourquoi le passage à un sous-programme d'un élément non défini d'un hachage le crée du même coup ?
- Comment faire l'équivalent en Perl d'une structure en C, d'une classe/d'un hachage en C++ ou d'un tableau de hachages ou de tableaux ?
- Comment utiliser une référence comme clef d'une table de hachage ?
- Comment manipuler proprement des données binaires ?
- Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ?
- Comment conserver des données persistantes entre divers appels de programme ?
- Comment afficher ou copier une structure de données récursive ?
- Comment définir des méthodes pour toutes les classes ou tous les objets ?
- Comment vérifier la somme de contrôle d'une carte de crédit ?
- Comment compacter des tableaux de nombres à virgule flottante simples ou doubles pour le code XS ?

**perlfaq5: Fichiers et formats.**

E/S (Entrées et Sorties) et autres éléments connexes : descripteurs de fichiers, vidage de tampons, formats d'écriture et mise en page.

- Comment vider ou désactiver les tampons en sortie ? Pourquoi m'en soucier ?
- Comment changer une ligne, effacer une ligne, insérer une ligne au milieu ou ajouter une ligne en tête d'un fichier ?
- Comment déterminer le nombre de lignes d'un fichier ?
- Comment utiliser l'option `-i` de Perl depuis l'intérieur d'un programme ?
- Comment puis-je copier un fichier ?
- Comment créer un fichier temporaire ?
- Comment manipuler un fichier avec des enregistrements de longueur fixe ?
- Comment rendre un descripteur de fichier local à une routine ? Comment passer des descripteurs à d'autres routines ? Comment construire un tableau de descripteurs ?
- Comment utiliser un descripteur de fichier indirectement ?
- Comment mettre en place un pied-de-page avec `write()` ?
- Comment rediriger un `write()` dans une chaîne ?
- Comment afficher mes nombres avec des virgules pour délimiter les milliers ?
- Comment traduire les tildes (`~`) dans un nom de fichier ?
- Pourquoi les fichiers que j'ouvre en lecture-écriture se voient-ils effacés ?
- Pourquoi `<*>` donne de temps en temps l'erreur "Argument list too long" ?
- Y a-t-il une fuite / un bug avec `glob()` ?
- Comment ouvrir un fichier dont le nom commence par ">" ou se termine par des espaces ?
- Comment renommer un fichier de façon sûre ?
- Comment verrouiller un fichier ?
- Pourquoi ne pas faire simplement `open(FH, ">file.lock")` ?
- Je ne comprends toujours pas le verrouillage. Je veux seulement incrémenter un compteur dans un fichier. Comment faire ?
- Je souhaite juste ajouter un peu de texte à la fin d'un fichier. Dois-je tout de même utiliser le verrouillage ?
- Comment modifier un fichier binaire directement ?
- Comment récupérer la date d'un fichier en perl ?
- Comment modifier la date d'un fichier en perl ?
- Comment écrire dans plusieurs fichiers simultanément ?
- Comment lire le contenu d'un fichier d'un seul coup ?
- Comment lire un fichier paragraphe par paragraphe ?
- Comment lire un seul caractère d'un fichier ? Et du clavier ?
- Comment savoir si un caractère est disponible sur un descripteur de fichier ?
- Comment écrire un `tail -f` en perl ?
- Comment faire un `dup()` sur un descripteur en Perl ?
- Comment fermer un descripteur connu par son numéro ?
- Pourquoi "C:\temp\foo" n'indique pas un fichier DOS ? Et même "C:\temp\foo.exe" ne marche pas ?
- Pourquoi `glob("*.*)` ne donne-t-il pas tous les fichiers ?
- Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi `-i` écrit-il dans des fichiers protégés ? N'est-ce pas un bug de Perl ?
- Comment sélectionner une ligne au hasard dans un fichier ?
- Pourquoi obtient-on des espaces étranges lorsqu'on affiche un tableau de lignes ?

**perlfaq6: Expressions rationnelles.**

Cette partie de la FAQ est incroyablement courte car les autres parties sont parsemées de réponses concernant les expressions rationnelles. Par exemple, décoder une URL ou vérifier si quelque chose est un nombre ou non relève du domaine des expressions rationnelles, mais ces réponses se trouvent ailleurs que dans ce document (dans *perlfaq9* « Comment décoder ou créer ces %-encodings sur le web ? » et dans *perlfaq4* « Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ? »).

- Comment utiliser les expressions rationnelles sans créer du code illisible et difficile à maintenir ?
- J'ai des problèmes pour faire une reconnaissance sur plusieurs lignes. Qu'est-ce qui ne va pas ?
- Comment extraire des lignes entre deux motifs qui sont chacun sur des lignes différentes ?
- J'ai mis une expression rationnelle dans `$/` mais cela ne marche pas. Qu'est-ce qui est faux ?
- Comment faire une substitution indépendante de la casse de la partie gauche mais préservant cette casse dans la partie droite ?
- Comment faire pour que `\w` reconnaisse les caractères nationaux ?
- Comment reconnaître une version locale-équivalente de `/[a-zA-Z]/` ?
- Comment protéger une variable pour l'utiliser dans une expression rationnelle ?



- À quoi sert vraiment /o ?
- Comment utiliser une expression rationnelle pour enlever les commentaires de type C d'un fichier ?
- Est-ce possible d'utiliser les expressions rationnelles de Perl pour reconnaître du texte bien équilibré ?
- Que veut dire "les expressions rationnelles sont gourmandes" ? Comment puis-je le contourner ?
- Comment examiner chaque mot dans chaque ligne ?
- Comment afficher un rapport sur les fréquences de mots ou de lignes ?
- Comment faire une reconnaissance approximative ?
- Comment reconnaître efficacement plusieurs expressions rationnelles en même temps ?
- Pourquoi les recherches de limite de mot avec \b ne marchent pas pour moi ?
- Pourquoi l'utilisation de &&, \$', or \$' ralentit tant mon programme ?
- À quoi peut bien servir \G dans une expression rationnelle ?
- Les expressions rationnelles de Perl sont-elles AFD ou AFN ? Sont-elles conformes à POSIX ?
- Qu'est ce qui ne va pas avec l'utilisation de grep dans un contexte vide ?
- Comment reconnaître des chaînes avec des caractères multi-octets ?
- Comment rechercher un motif fourni par l'utilisateur ?

#### **perlfaq7: Questions générales sur le langage Perl.**

Traite des questions générales sur le langage Perl qui ne trouvent leur place dans aucune autre section.

- Puis-je avoir une BNF/yacc/RE pour le langage Perl ?
- Quels sont tous ces \$@%&\* de signes de ponctuation, et comment savoir quand les utiliser ?
- Dois-je toujours/jamais mettre mes chaînes entre guillemets ou utiliser les points-virgules et les virgules ?
- Comment ignorer certaines valeurs de retour ?
- Comment bloquer temporairement les avertissements ?
- Qu'est-ce qu'une extension ?
- Pourquoi les opérateurs de Perl ont-ils une précedence différente de celle des opérateurs en C ?
- Comment déclarer/créer une structure ?
- Comment créer un module ?
- Comment créer une classe ?
- Comment déterminer si une variable est souillée ?
- Qu'est-ce qu'une fermeture ?
- Qu'est-ce que le suicide de variable et comment le prévenir ?
- Comment passer/renvoyer {une fonction, un handle de fichier, un tableau, un hachage, une méthode, une expression rationnelle} ?
- Comment créer une variable statique ?
- Quelle est la différence entre la portée dynamique et lexicale (statique) ? Entre local() et my() ?
- Comment puis-je accéder à une variable dynamique lorsqu'un
- Quelle est la différence entre les liaisons profondes et superficielles ?
- Pourquoi "my(\$foo) = <FICHER>;" ne marche pas ?
- Comment redéfinir une fonction, un opérateur ou une méthode prédéfini ?
- Quelle est la différence entre l'appel d'une fonction par &foo et par foo() ?
- Comment créer une instruction switch ou case ?
- Comment intercepter les accès aux variables, aux fonctions, aux méthodes indéfinies ?
- Pourquoi une méthode incluse dans ce même fichier ne peut-elle pas être trouvée ?
- Comment déterminer mon paquetage courant ?
- Comment commenter un grand bloc de code perl ?
- Comment supprimer un paquetage ?
- Comment utiliser une variable comme nom de variable ?
- Que signifie "bad interpreter" ?

#### **perlfaq8: Interaction avec le système.**

Traite des questions concernant les interactions avec le système d'exploitation. Cela inclut les mécanismes de communication inter-processus (IPC – Inter Process Communication en anglais), le pilotage de l'interface utilisateur (clavier, écran et souris), et d'une façon générale tout ce qui ne relève pas de la manipulation de données.

- Comment savoir sur quel système d'exploitation je tourne ?
- Pourquoi ne revient-on pas après un exec() ?
- Comment utiliser le clavier/écran/souris de façon élaborée ?
- Comment afficher quelque chose en couleur ?
- Comment lire simplement une touche sans attendre un appui sur "entrée" ?
- Comment vérifier si des données sont en attente depuis le clavier ?
- Comment effacer l'écran ?
- Comment obtenir la taille de l'écran ?
- Comment demander un mot de passe à un utilisateur ?

- Comment lire et écrire sur le port série ?
- Comment décoder les fichiers de mots de passe cryptés ?
- Comment lancer un processus en arrière plan ?
- Comment capturer un caractère de contrôle, un signal ?
- Comment modifier le fichier masqué (shadow) de mots de passe sous Unix ?
- Comment positionner l'heure et la date ?
- Comment effectuer un sleep() ou alarm() de moins d'une seconde ?
- Comment mesurer un temps inférieur à une seconde ?
- Comment réaliser un atexit() ou setjmp()/longjmp() ? (traitement d'exceptions)
- Pourquoi mes programmes avec socket() ne marchent pas sous System V (Solaris) ? Que signifie le message d'erreur « Protocole non supporté » ?
- Comment appeler les fonctions C spécifiques à mon système depuis Perl ?
- Où trouver les fichiers d'inclusion pour ioctl() et syscall() ?
- Pourquoi les scripts perl en setuid se plaignent-ils d'un problème noyau ?
- Comment ouvrir un tube depuis et vers une commande simultanément ?
- Pourquoi ne puis-je pas obtenir la sortie d'une commande avec system() ?
- Comment capturer la sortie STDERR d'une commande externe ?
- Pourquoi open() ne retourne-t-il pas d'erreur lorsque l'ouverture du tube échoue ?
- L'utilisation des apostrophes inversées dans un contexte vide
- Comment utiliser des apostrophes inversées sans traitement du shell ?
- Pourquoi mon script ne lit-il plus rien de STDIN après que je lui ai envoyé EOF (^D sur Unix, ^Z sur MS-DOS) ?
- Comment convertir mon script shell en perl ?
- Puis-je utiliser perl pour lancer une session telnet ou ftp ?
- Comment écrire "expect" en Perl ?
- Peut-on cacher les arguments de perl sur la ligne de commande aux programmes comme "ps" ? =item \* J'ai {changé de répertoire, modifié mon environnement} dans un script perl. Pourquoi les changements disparaissent-ils lorsque le script se termine ? Comment rendre mes changements visibles ?
- Comment fermer le descripteur de fichier attaché à un processus sans attendre que ce dernier se termine ?
- Comment lancer un processus démon ?
- Comment savoir si je tourne de façon interactive ou pas ?
- Comment sortir d'un blocage sur événement lent ?
- Comment limiter le temps CPU ?
- Comment éviter les processus zombies sur un système Unix ?
- Comment utiliser une base de données SQL ?
- Comment terminer un appel à system() avec un control-C ?
- Comment ouvrir un fichier sans bloquer ?
- Comment faire la différence entre les erreurs shell et les erreurs perl ?
- Comment installer un module du CPAN ?
- Quelle est la différence entre require et use ?
- Comment gérer mon propre répertoire de modules/bibliothèques ?
- Comment ajouter le répertoire dans lequel se trouve mon programme dans le chemin de recherche des modules / bibliothèques ?
- Comment ajouter un répertoire dans mon chemin de recherche (@INC) à l'exécution ?
- Qu'est-ce que socket.ph et où l'obtenir ?

### **perlfaq9: Réseau.**

Traite des questions relatives aux aspects réseau, à internet et un peu au web.

- Quelle est la forme correcte d'une réponse d'un script CGI ?
- Mon script CGI fonctionne en ligne de commandes mais pas depuis un navigateur. (500 Server Error)
- Comment faire pour obtenir de meilleurs messages d'erreur d'un programme CGI ?
- Comment enlever les balises HTML d'une chaîne ?
- Comment extraire des URL ?
- Comment télécharger un fichier depuis la machine d'un utilisateur ? Comment ouvrir un fichier d'une autre machine ?
- Comment faire un menu pop-up en HTML ?
- Comment récupérer un fichier HTML ?
- Comment automatiser la soumission d'un formulaire HTML ?
- Comment décoder ou créer ces %-encodings sur le web ?
- Comment rediriger le navigateur vers une autre page ?
- Comment mettre un mot de passe sur mes pages Web ?
- Comment éditer mes fichiers .htpasswd et .htgroup en Perl ?

- Comment être sûr que les utilisateurs ne peuvent pas entrer de valeurs dans un formulaire qui font faire de vilaines choses à mon script CGI ?
- Comment analyser un en-tête de mail ?
- Comment décoder un formulaire CGI ?
- Comment vérifier la validité d'une adresse électronique ?
- Comment décoder une chaîne MIME/BASE64 ?
- Comment renvoyer l'adresse électronique de l'utilisateur ?
- Comment envoyer un mail ?
- Comment utiliser MIME pour attacher des documents à un mail ?
- Comment lire du courrier ?
- Comment trouver mon nom de machine / nom de domaine / mon adresse IP ?
- Comment récupérer un article de news ou les groupes actifs ?
- Comment récupérer/envoyer un fichier par FTP ?
- Comment faire du RPC en Perl ?

## 17.6 TRADUCTION

### 17.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 17.6.2 Traducteur

Marc Carmier <carmier@immortels.frmug.org>. Paul Gaborit <paul.gaborit at enstimac.fr> (mise à jour).

### 17.6.3 Relecture

Pascal Ethvignot <pascal@encelade.frmug.org>

# Chapitre 18

## perlfaq1

Questions d'ordre général sur Perl

### 18.1 DESCRIPTION

Cette section de la FAQ répond aux questions d'ordre général et de haut niveau sur Perl.

#### 18.1.1 Qu'est ce que Perl ?

Perl est un langage de programmation de haut niveau avec un héritage éclectique écrit par Larry Wall et un bon millier de développeurs. Il dérive de l'omniprésent langage C et, dans une moindre mesure, de Sed, Awk, du shell Unix et d'au moins une douzaine d'autres langages et outils. Son aisance à manipuler les processus, les fichiers et le texte le rend particulièrement bien adapté aux tâches faisant intervenir le prototypage rapide, les utilitaires système, les outils logiciels, les gestionnaires de tâches, l'accès aux bases de données, la programmation graphique, les réseaux, et la programmation web. Ces points forts en font un langage particulièrement populaire auprès des administrateurs système et des auteurs de scripts CGI, mais l'utilisent également des mathématiciens, des généticiens, des journalistes et même des managers. Et peut être vous aussi ?

#### 18.1.2 Qui supporte Perl ? Qui le développe ? Pourquoi est-il gratuit ?

La culture d'origine d'Internet et les croyances profondément ancrées de l'auteur de Perl, Larry Wall, ont donné naissance à la politique de distribution gratuite et ouverte de perl. Perl est soutenu par ses utilisateurs. Le noyau, les bibliothèques standards Perl, les modules optionnels, et la documentation que vous êtes en train de lire ont tous été rédigés par des volontaires. Vous pouvez consulter les notes personnelles à la fin du fichier README de la distribution du code source de Perl pour plus de détails. L'historique des versions de Perl (jusqu'à la 5.005) est disponible ici : *perlhists*.

En particulier, l'équipe principale de développement (connue sous le nom 'Perl Porters') est une bande hétéroclite d'individus des plus altruistes engagés à faire un logiciel gratuit meilleur que tout ce que vous pourriez trouver dans le commerce. Vous pouvez glaner des informations sur les développements en cours en lisant les archives sur <http://www.xray.mpe.mpg.de/mailling-lists/perl5-porters/> et <http://archive.developer.com/perl5-porters@perl.org/> ou via la passerelle de news nntp [nntp://nntp.perl.org/perl.perl5.porters](http://nntp.perl.org/perl.perl5.porters) ou son interface web <http://nntp.perl.org/group/perl.perl5.porters> ou encore en lisant la faq sur <http://simon-cozens.org/writings/p5p-faq>. Vous pouvez aussi vous inscrire à la liste de diffusion en envoyant une demande d'inscription à [perl5-porters-request@perl.org](mailto:perl5-porters-request@perl.org) (un message vide sans sujet suffit).

Bien que le projet GNU inclue Perl dans ses distributions, il n'y a pas à proprement parler de "GNU Perl". Perl n'est pas produit ni entretenu par la Free Software Foundation. Les termes de la licence de Perl sont aussi moins restrictifs que ceux de la licence GNU.

Vous pouvez obtenir une assistance commerciale pour Perl si vous le souhaitez, mais pour la plupart des utilisateurs, une assistance informelle devrait être largement suffisante. Regardez la réponse à question "Où acheter une version commerciale de perl" pour de plus amples informations.

### 18.1.3 Quelle version de Perl dois-je utiliser ?

(contribution de brian d foy)

Tous les goûts sont dans la nature... il n'y a donc pas de réponse à cette question qui puisse convenir à tout le monde. En général, vous voudrez utiliser soit la dernière version stable soit celle qui la précède immédiatement. Actuellement, ce sont respectivement perl5.8.x et perl5.6.x.

Après cela, vous devez tout de même considérer plusieurs pour décider au mieux de vos intérêts.

- Si tout marche bien, la mise à jour de perl peut faire que ça ne marche plus (ou au minimum amener de nouveaux avertissements).
- Les dernières versions de perl bénéficie de plus de corrections de bugs.
- La communauté Perl fournit de l'aide plutôt pour les versions les plus récentes, il vous sera donc plus facile de vous faire aider avec ces versions.
- Les versions antérieures à perl5.004 présentent de très sérieux problèmes de sécurité comme des dépassements de tampons et font l'objet de plusieurs avertissements du CERT (par exemple, <http://www.cert.org/advisories/CA-1997-17.html>).
- Les toutes dernières versions sont sans aucun doute les moins déployées et testées donc, si vous n'aimez pas prendre de risques, vous pouvez attendre quelques mois après leur sortie afin de voir les problèmes rencontrés par les autres.
- L'avant-dernière version (actuellement perl5.6.x) est habituellement maintenue assez longtemps, bien qu'elle ne soit pas au même niveau que la version courante.
- Personne ne met à jour perl4.x. Il y a cinq ans, c'était déjà une simple carcasse de chameau mort. Maintenant, ce n'est même plus un squelette et ses os d'un blanc immaculé s'érodent et tombent en miettes.
- perl6.x ne verra pas le jour tout de suite. Restez attentif mais ne craignez pas d'avoir à changer de versions majeures de Perl avant un moment (NdT: pas avant fin 2006).
- Il existe deux pistes parallèles de développement de perl : une piste pour les versions de maintenance et une piste pour les versions expérimentales. Les versions de maintenance sont stables et leur numéro de version mineur est paire (par exemple perl5.8.x où 8 est le numéro de version mineur). Les versions expérimentales peuvent inclure des fonctionnalités qui ne sont pas dans les versions stables et leur numéro de version mineur est impair (par exemple perl5.9.x où 9 est le numéro de version mineur).

### 18.1.4 Qu'est-ce que veut dire perl4, perl5 ou perl6 ?

(contribution de brian d foy)

Pour faire court, perl4 est le passé, perl5 est le présent et perl6 est le futur.

Le numéro après perl (le 5 dans perl5) est le numéro de version majeur de l'interpréteur perl ainsi que la version du langage. Chaque version majeure possède des différences significatives que les versions antérieures ne peuvent pas supporter.

La version majeure courante de Perl est perl5 et date de 1994. Elle peut exécuter des scripts de la version majeure précédente (perl4 qui date de mars 1991 mais a des différences importantes. Elle introduit le concept de références, les structures de données complexes et les modules. L'interpréteur perl5 est une réécriture complète des sources perl précédents.

Perl6 est la prochaine version majeure de Perl mais il est encore en développement tant du point de vue de sa syntaxe que du point de vue des choix de conception. Le travail a commencé en 2002 et est encore en cours. La plupart des nouvelles fonctionnalités sont apparues dans les dernières versions de perl5 et certains modules perl5 vous permettent même d'utiliser quelques éléments de la syntaxe perl6 dans vos programmes. Vous pouvez en apprendre plus sur <http://dev.perl.org/perl6/>.

Voir *perlhst* pour un historique des révisions de Perl.

### 18.1.5 Qu'est-ce que Ponie ?

Lors de la convention O'Reilly Open Source Software en 2003, Artur Bergman, Fotango et la Fondation Perl ont annoncé un projet nommé Ponie dont le but est de faire tourner perl5 sur la machine virtuelle Parrot. Ponie signifie Perl On New Internal Engine (Perl sur le nouveau moteur). L'implémentation du langage Perl 5.10 sera utilisée pour Ponie et il n'y aura aucune différence de langage entre perl5 et ponie. Ponie n'est pas une réécriture complète de perl5.

### 18.1.6 Qu'est-ce que perl6 ?

Lors de la seconde convention O'Reilly Open Source Software, Larry Wall a annoncé que le développement de Perl6 allait commencé. Perl6 avait parfois été utilisé pour désigner le projet Topaz de Chip Salzenberg visant à réécrire Perl en C++. Ce projet, bien qu'ayant fourni des apports non négligeables pour la prochaine version de Perl et de son implémentation, est maintenant abandonné.

Si vous voulez en savoir plus sur Perl6 ou si vous souhaitez aider à améliorer Perl, allez sur la page des développeurs de Perl6 <http://dev.perl.org/perl6/>.

Il n'y a pas encore de date prévue pour Perl6 et même après sa sortie, Perl5 restera encore longtemps maintenu. N'attendez donc pas Perl6 pour faire ce que vous devez faire.

"Sérieusement, nous réinventons tout ce qui doit être réinventé." – Larry Wall

### 18.1.7 Est-ce que Perl est stable ?

Les nouvelles versions produites, qui incluent des corrections de bugs et de nouvelles fonctionnalités, sont amplement testées avant d'être distribuées. Depuis la version 5.000, on compte en moyenne une version par an.

Larry et l'équipe de développement Perl font occasionnellement des changements dans le noyau interne du langage, mais tous les efforts possibles sont déployés pour assurer la compatibilité descendante. Bien que quelques scripts perl4 ne tournent pas impeccablement sous perl5, une mise à jour de perl ne devrait presque jamais invalider un programme écrit pour une version plus ancienne. (exception faite des corrections de bugs accidentels et des rares nouveaux mots réservés).

### 18.1.8 Est-il difficile d'apprendre Perl ?

Non, il est facile de débiter et même de continuer l'apprentissage de Perl. Il ressemble à la plupart des langages de programmation que vous avez probablement rencontrés ; aussi, si vous avez déjà écrit un programme en C, un script awk, un script shell ou même un programme en BASIC, vous avez déjà fait une partie du chemin.

La plupart des tâches ne requiert qu'un petit sous-ensemble du langage Perl. Une des idées phares en matière de développement Perl est : "Il y a plus d'une façon de procéder". La courbe d'apprentissage de Perl est étroite (facile à apprendre) et longue (il y a beaucoup de choses faisables si vous le voulez réellement).

Enfin, puisque Perl est souvent (mais pas toujours, et certainement pas par définition) un langage interprété, vous pouvez écrire vos programmes et les tester sans phase intermédiaire de compilation, vous permettant ainsi d'expérimenter et de déboguer/tester rapidement et facilement. Cette facilité d'expérimentation aplatit encore plus sa courbe d'apprentissage.

Les choses qui facilitent l'apprentissage de Perl sont : l'expérience d'Unix, quasiment n'importe quelle expérience de la programmation, une compréhension des expressions rationnelles, et la capacité de comprendre le code des autres. S'il y a quelque chose que vous avez besoin de faire, elle a probablement été déjà faite, et un exemple fonctionnant est généralement disponible gratuitement. N'oubliez pas non plus les modules perl. Ils sont abordés dans la partie 3 de cette FAQ, et avec le CPAN dans la partie 2.

### 18.1.9 Est-ce que Perl tient la comparaison avec d'autres langages comme Java, Python, REXX, Scheme ou Tcl ?

Oui pour certains domaines, moins pour d'autres. Définir précisément dans quels domaines Perl est bon ou mauvais est souvent une question de choix personnel, et poser cette question sur Usenet risque fortement de déclencher un débat houleux et stérile.

La meilleure chose à faire est certainement d'essayer d'écrire le code équivalent dans plusieurs langages pour accomplir un ensemble de tâches. Ces langages ont leurs propres newgroups dans lesquels vous pouvez en apprendre plus (et non, espérons le, vous disputer) sur eux.

Des comparatifs se trouvent sur <http://www.perl.com/do@FMTEYEWTK/versus/> si vous ne pouvez vraiment pas vous en passer.

### 18.1.10 Que puis-je faire avec Perl ?

Perl est suffisamment souple et extensible pour que vous puissiez l'employer virtuellement pour tout type de tâche, du traitement de fichier ligne par ligne à des grands systèmes élaborés. Pour de nombreuses personnes, Perl remplace avantageusement les outils existants pour les scripts shell. Pour d'autres, c'est un substitut efficace et de haut niveau pour tout ce qu'ils programmeraient en langage de bas niveau comme C ou C++. En définitive c'est à vous (et certainement à votre direction...) de voir pour quelles tâches vous allez utiliser Perl ou non.

Si vous avez une bibliothèque fournissant une API, vous pouvez en rendre n'importe quel composant disponible comme une fonction ou une variable Perl supplémentaire en utilisant une extension Perl écrite en C ou C++, et dynamiquement liée dans votre interpréteur perl principal. À l'opposé, vous pouvez également écrire votre programme principal en C/C++, et ensuite lier du code Perl au vol pour créer une puissante application. Voir *perlembed*.

Ceci dit, il y aura toujours des langages dédiés à une classe de problèmes qui sont tout simplement plus pratiques. Perl essaye d'être tout à la fois pour tout le monde, mais rien de précis pour personne. Les exemples de langages spécialisés qui viennent à l'esprit comptent prolog et matlab.

### 18.1.11 Quand ne devrais-je pas programmer en Perl ?

Quand votre patron vous l'interdit - mais envisagez de lui trouver un remplaçant :-)

En fait, une bonne raison pour ne pas utiliser Perl est d'avoir une application déjà existante écrite dans un autre langage qui est toute faite (et bien faite), ou si vous avez une application conçue pour une tâche spécifique dans un langage particulier (i.e. prolog, make)

Pour des raisons variées, Perl n'est probablement pas bien adapté pour des systèmes embarqués temps-réel, des développements systèmes de bas niveau, des travaux comme des pilotes de périphérique ou du code à commutation de contexte, des applications parallèles complexes en mémoire partagée, ou des applications très grandes. Vous remarquerez que perl n'est pas lui-même écrit en Perl.

Le nouveau compilateur Perl de code natif peut éventuellement réduire ces limitations jusqu'à un certain degré, mais comprenez que Perl reste fondamentalement un langage à typage dynamique, et non à typage statique. On ne vous en voudra pas si vous ne lui faites pas confiance pour un code de centrale nucléaire ou de contrôle de chirurgie cérébrale. Larry en dormira d'autant mieux - en ne tenant pas compte des programmes de Wall Street. :-)

### 18.1.12 Quelle est la différence entre "perl" et "Perl" ?

Un bit. Ah, mais vous ne parlez pas de codes ASCII ? :-) Larry emploie le terme "Perl" pour désigner le langage en lui-même, tandis que "perl" désigne son implémentation, c'est-à-dire l'interpréteur actuel. D'où la boutade de Tom : "Rien d'autre que perl ne peut analyser Perl". Vous pouvez ou non choisir de suivre cet usage. Le parallélisme implique par exemple que "awk et perl" et "Python et Perl" ont l'air correct, tandis que "awk et Perl" ou "Python et perl" ne le sont pas. Mais n'écrivez jamais "PERL" parce que perl n'est pas un acronyme, si l'on ne tient pas compte du folklore apocryphe ni des expansions a posteriori.

### 18.1.13 Parle-t-on de programme Perl ou de script Perl ?

Larry ne s'en soucie pas vraiment. Il dit (à moitié pour rire) qu'un "script est ce que l'on donne aux acteurs. Un programme est ce que vous donnez à l'audience."

Originellement, un script était une séquence en boîte de commandes interactives normales, c'est-à-dire un script de chat. Une chose telle qu'un script de chat UUCP ou PPP ou un script expect est plutôt conforme à cette description, tout comme les scripts de configuration exécutés par un programme lors de son lancement, comme *.cshrc* ou *.ircrc*, par exemple. Les scripts de chat étaient juste des pilotes pour des programmes existants, et non pas des programmes indépendants.

Un scientifique de l'informatique expliquera convenablement que tous les programmes sont interprétés, et que la seule question qui se pose est à quel niveau. Mais si vous posez cette question à quelqu'un d'autre, il pourra vous dire qu'un *programme* a été compilé une seule fois pour devenir du code machine physique, et peut alors être exécuté plusieurs fois, tandis qu'un *script* doit être traduit par un programme à chaque fois qu'il est utilisé.

Les programmes Perl ne sont (habituellement) ni strictement compilés, ni strictement interprétés. Ils peuvent être compilés sous la forme d'un pseudo-code (pour une sorte de machine virtuelle Perl) ou dans un tout autre langage, comme du C ou de l'assembleur. Vous ne pouvez pas dire juste en le regardant si la source est destinée à un interpréteur pur, un interpréteur d'arbre syntaxique, un interpréteur de pseudo-code, ou un compilateur générant du code machine, donc il est délicat ici de donner une réponse définitive.

Maintenant que "script" et "scripting" sont des termes ayant été abusés par des marketeux sans scrupules ou ignorants pour leur propre bénéfice infâme, ils ont commencé à avoir des significations étranges et souvent péjoratives, comme "pas sérieux", ou "pas de la vraie programmation". Par conséquent, certains programmeurs Perl préfèrent totalement les éviter.

### 18.1.14 Qu'est ce qu'un JAPH ?

Ce sont les signatures "just another perl hacker" que certains utilisent pour signer leurs messages. Randal Schwartz les a rendues célèbres. Environ une centaine parmi les plus anciennes sont disponibles sur <http://www.cpan.com/CPAN/misc/japh>.

### 18.1.15 Où peut on trouver une liste des mots d'esprit de Larry Wall ?

Plus d'une centaine de boutades de Larry, issues de ses messages ou du code source, peuvent être trouvées à l'adresse <http://www.cpan.org/mis@lwall-quotes.txt.gz>.

### 18.1.16 Comment convaincre mon administrateur système/chef de projet/employés d'utiliser Perl/Perl5 plutôt qu'un autre langage ?

Si votre directeur ou vos employés se méfient des logiciels non entretenus, ou non officiellement fournis avec votre système d'exploitation, vous pouvez essayer d'en appeler à leur intérêt personnel. Si les programmeurs peuvent être plus productifs en utilisant les constructions, la fonctionnalité, la simplicité et la puissance de Perl, alors le directeur/chef de projet/employé type devrait être convaincu. Quant à l'usage de Perl en général, il est parfois aussi utile pour démontrer que les temps de livraison peuvent être réduits en l'utilisant plutôt qu'un autre langage.

Si vous avez un projet avec des impératifs, notamment en termes de portabilité ou de tests, Perl devrait certainement apporter une solution rapide et viable. En plus de tout effort de persuasion, vous ne devriez pas manquer de montrer que Perl est utilisé, assez intensivement et avec des résultats solides et fiables, dans de nombreuses grandes entreprises informatiques de logiciel et de matériel à travers le monde. En fait, de nombreuses distributions Unix livrent maintenant Perl en standard. Les forum de discussion usenet apportent l'assistance nécessaire si vous ne trouvez pas la réponse dans la documentation *complète*, y compris cette FAQ.

Voir <http://www.perl.org/advocacy/> pour plus d'informations.

Si vous êtes confronté à des réticences pour mettre à jour une version antérieure de perl, insistez sur le fait que la version 4 n'est plus entretenue ni suivie par l'équipe de développement Perl. Un autre argument de taille en faveur de Perl5 est le grand nombre de modules et d'extensions qui réduisent considérablement le temps de développement pour tout type de tâche. Mentionnez également que la différence entre la version 4 et la version 5 de Perl est similaire à celle entre awk et C++. (Bon d'accord, ce n'est peut-être pas si différent, mais l'idée est là.) Si vous voulez de l'assistance et des garanties raisonnables que ce que vous développez fonctionnera encore dans le futur, alors vous devriez utiliser une version maintenue. En décembre 2003, cela signifie utiliser soit la 5.8.2 (sortie en novembre 2003) ou l'une des versions plus anciennes telles que la 5.6.2 (sortie aussi en novembre 2003 pour assurer la compilation de perl 5.6 sur les machines récentes puisque la 5.6.1 datait d'avril 2001) ou la 5.005\_03 (qui date de mars 1999). À la rigueur la 5.004\_05 si vous avez **absolument** besoin d'une aussi vieille version (avril 1999) pour des raisons de stabilité. Aucune version antérieure ne devrait être utilisée.

À noter en particulier la chasse de grande envergure aux erreurs de dépassement de tampon survenues dans la version 5.004. Toute version antérieure à celle-là, y compris perl4, est considérée comme non sûre et devrait être mise à jour aussitôt que possible.

En août 2000 dans toutes les distributions Linux, il a été trouvé un nouveau problème de sécurité dans la commande optionnelle 'suidperl' (qui n'est pas installée par défaut) fournie avec les branches Perl 5.6, 5.005 et 5.004 (voir <http://www.cpan.org/src/5.0/sperl-2000-08-05/>). Les versions de maintenance 5.6.1 et 5.8.0 comblent ce trou de sécurité. La plupart, mais pas toutes, des distributions Linux proposent des correctifs pour cette vulnérabilité (voir <http://www.linuxsecurity.com/advisories/>) mais la bonne méthode consiste tout de même à passer au moins à Perl 5.6.1.

## 18.2 AUTEUR ET COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen, Nathan Torkington et les autres auteurs cités. Tous droits réservés.

Cette documentation est libre. Vous pouvez la redistribuer ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendamment de sa distribution, le code de tous les exemples est ici du domaine public. Vous êtes autorisé et même encouragé à utiliser ou modifier ce code pour vos propres programmes, que ce soit pour le plaisir ou à but lucratif. Un simple commentaire dans le code remerciant la FAQ serait courtois, quoique non exigé.



## 18.3 TRADUCTION

### 18.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 18.3.2 Traducteur

Traduction initiale : Emmanuel BOURG <smanux@dream.club-internet.fr>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit <paul.gaborit@enstimac.fr>.

### 18.3.3 Relecture

Gérard Delafond.

# Chapitre 19

## perlfaq2

Trouver et apprendre Perl

### 19.1 DESCRIPTION

Cette section de la FAQ vous permettra de trouver les sources et la documentation de Perl ainsi que de l'aide et autres choses similaires.

#### 19.1.1 Quelles machines supportent perl ? Où puis-je trouver Perl ?

La version standard de perl (celle maintenue par l'équipe de développement de perl) est distribuée uniquement sous forme de code source. Vous pouvez la trouver sur <http://www.cpan.com/src/latest.tar.gz> sous un format standard d'Internet (une archive gzippée au format tar POSIX).

Perl se compile et fonctionne sur un nombre ahurissant de plates-formes. Quasiment tous les dérivés actuels et connus d'Unix sont supportés ainsi que d'autres systèmes comme VMS, DOS, OS/2, Windows, QNX, BeOS, OS X, MPE/iX et Amiga.

Des distributions binaires pour quelques plates-formes propriétaires, y compris les systèmes Apple, sont disponibles dans le répertoire <http://www.cpan.com/ports/>. Comme elles ne font pas partie de la distribution standard, elles peuvent différer (et diffèrent de fait) de la version de base de perl sur plusieurs points. Pour connaître les différences, vous devrez lire leurs notes respectives d'exploitation (release notes). Ces différences peuvent être des plus (par ex. des extensions pour des fonctionnalités spécifiques à une plate-forme qui ne sont pas supportées par la version source de perl) ou des moins (par ex. peuvent être basées sur de vieilles versions des sources perl).

#### 19.1.2 Comment trouver une version binaire de perl ?

Si vous n'avez pas de compilateur C parce que par exemple votre vendeur n'en fournit pas sur votre système, la meilleure chose à faire est de récupérer une version binaire de gcc depuis Internet et de l'utiliser pour compiler perl. CPAN ne fournit de versions binaires que pour les systèmes pour lesquels il est très difficile de trouver des compilateurs gratuits, mais pas pour les systèmes Unix.

Quelques URL qui devraient vous aider :

<http://www.cpan.org/ports/>  
<http://www.perl.com/pub/language/info/software.html>

Quelqu'un cherchant un perl pour Win16 peut s'intéresser au portage djgpp de Laszlo Molnar sur <http://www.cpan.com/ports/msdos/>, qui est fourni avec des instructions d'installation claires. Un guide d'installation simple pour MS-DOS utilisant le portage OS/2 d'Ilya Zakharevich est disponible sur <http://www.cs.ruu.nl/%7Epiet/perl5dos.html> et de façon similaire pour Windows 3.1 sur <http://www.cs.ruu.nl/%7Epiet/perlwin3.html>.

### 19.1.3 Je n'ai pas de compilateur C sur mon système. Comment puis-je compiler mon propre interpréteur Perl ?

Si vous n'avez pas de compilateur C, vous êtes maudits et votre vendeur devrait être sacrifié aux dieux du soleil (Sun). Mais cela ne vous aidera pas beaucoup.

Il vous faut trouver une distribution binaire de gcc pour votre système. Pour savoir où la trouver, consultez les FAQs Usenet concernant votre système d'exploitation.

### 19.1.4 J'ai copié le binaire perl d'une machine sur une autre mais les scripts ne fonctionnent pas.

Vous avez probablement oublié les bibliothèques ou alors les chemins d'accès différent. Vous devriez plutôt installer toute la distribution sur la machine finale et tapez `make install`. La plupart des autres méthodes sont vouées à l'échec.

Pour vérifier que tout est installé au bon endroit, il suffit d'afficher la valeur hard-codée de `@INC` que perl utilise :

```
% perl -le 'print for @INC'
```

Si cette commande liste des répertoires qui n'existent pas sur votre système, vous devrez les créer et y placer les bibliothèques ou alors y créer des liens symboliques, des alias ou des raccourcis de manière appropriée. `@INC` est aussi affiché lors de la sortie de :

```
% perl -V
```

Vous pouvez aussi consulter Comment gérer mon propre répertoire de modules/bibliothèques ? in *perlfaq8*.

### 19.1.5 J'ai récupéré les sources et j'essaie de les compiler mais gdbm/dynamic loading/malloc/linking/... échoue. Comment faire pour que ça marche ?

Lisez le fichier *INSTALL* qui est dans la distribution source. Il décrit en détail comment se débrouiller avec la plupart des particularités d'un système ou d'une architecture pour lesquelles le script *Configure* ne peut pas s'en sortir.

### 19.1.6 Quels modules et extensions existent pour Perl ? Qu'est-ce que CPAN ? Que signifie CPAN/src/... ?

CPAN signifie Comprehensive Perl Archive Network (Réseau d'archive exhaustive sur Perl). C'est une archive d'environ 1.2Go dupliquée sur presque 200 machines à travers le monde. CPAN contient du code source, des portages non natifs, des documentations, des scripts et plein de modules et/ou extensions de tierces parties adaptés à tout depuis des interfaces pour des bases de données commerciales jusqu'au contrôle fin du clavier ou de l'écran en passant par des scripts CGI ou d'exploration du WEB. La serveur web de référence pour CPAN est <http://www.cpan.org/> et il existe un multiplexeur CPAN sur <http://www.cpan.org/CPAN.html> qui choisit le miroir le plus proche de vous via le DNS. Voir <http://www.cpan.org/CPAN> (sans le `sh;ash` final) pour comprendre comment ça fonctionne. Il existe aussi <http://mirror.cpan.org/>, une jolie interface pour accéder au répertoire des miroirs <http://www.cpan.org/MIRRORED.BY>.

Lisez la FAQ CPAN sur <http://www.cpan.org/misc/cpan-faq.html> pour trouver les réponses aux questions les plus fréquentes concernant CPAN et en particulier pour savoir comment devenir un miroir.

CPAN/chemin/... est une convention pour nommer des fichiers disponibles sur les sites CPAN. CPAN désigne le répertoire de base d'un miroir CPAN et le reste du chemin est celui depuis ce répertoire jusqu'au fichier lui-même. Par exemple, si vous utilisez <ftp://ftp.funet.fi/pub/languages/perl/CPAN> comme site CPAN, le fichier CPAN/misc/japh est téléchargeable à <ftp://ftp.funet.fi/pub/languages/perl/CPAN/misc/japh>.

Sachant qu'il existe plusieurs milliers de modules dans l'archive, il en existe certainement un qui fait presque ce à quoi vous pensez. Les catégories actuelles disponibles dans CPAN/modules/by-category/ sont entre autres : Perl core modules (modules Perl de base) ; development support (aide au développement) ; operating system interfaces (interfaces avec le système d'exploitation) ; networking, devices, and interprocess communication (réseaux, devices et communication inter-process) ; data type utilities (utilitaires de type de donnée) ; database interfaces (interfaces avec les bases de données) ; user interfaces (interfaces utilisateur) ; interfaces to other languages (interfaces avec d'autres langages) ; filenames, file systems, and file locking (noms de fichiers, systèmes de fichiers et verrouillage de fichiers) ; internationalization and locale (internationalisation et locale) ; world wide web support (aide pour le WEB) ; server and daemon utilities (utilitaires serveur et démon) ; archiving and compression (archivages et compression) ; image manipulation (traitement d'images) ; mail and news (messagerie électronique et groupe de discussion) ; control flow utilities (utilitaires de contrôle de flux) ; filehandle and I/O (filehandle et entrée/sortie (I/O)) ; Microsoft Windows modules (modules Microsoft Windows) ; miscellaneous modules (modules divers).

Voir <http://www.cpan.org/modules/00modlist.long.html> ou <http://search.cpan.org/> pour une liste de modules plus complètes et classée par catégorie.

### 19.1.7 Existe-t-il une version de Perl certifiée ISO ou ANSI ?

Sûrement pas. Larry estime qu'il sera déclaré malade (NDT : jeu de mot sur certified) avant que Perl ne le soit.

### 19.1.8 Où puis-je trouver des informations sur Perl ?

La documentation complète de Perl est contenue dans la distribution de Perl. Si vous avez installé Perl, vous avez probablement installé la documentation en même temps : tapez `man perl` si vous êtes sur un système ressemblant à Unix. Cela vous amènera vers d'autres pages importantes du manuel et entre autres celles indiquant comment fixer votre variable `$MANPATH`. Si vous n'êtes pas sur un système Unix, l'accès pourra être différent. Il est possible, par exemple, que la documentation n'existe qu'au format HTML. Mais en tout état de cause, toute distribution Perl correcte fournit une documentation complète.

Vous pouvez aussi essayer `perldoc perl` si votre système ne possède pas de commande `man` ou si elle est mal installée. Si ça ne fonctionne pas, regardez dans `/usr/local/lib/perl5/pod` pour trouver de la documentation.

Si tout cela échoue, consultez <http://perldoc.perl.org/> qui propose la documentation complète aux formats HTML et PDF.

De nombreux ouvrages de qualité existent pour Perl – voir plus loin pour de plus amples informations.

Des tutoriels sont inclus dans les versions actuelles ou à venir de Perl. Par exemple *perltoot* pour les objets, *perlboot* pour ceux qui débutent totalement en approche objet, *perlopentut* pour la sémantique de l'ouverture de fichiers, *perlrefut* pour gérer les références, *perlretut* pour les expressions rationnelles (ou régulières), *perlthrtut* pour les fils d'exécution (les threads), *perldebtut* pour le débogage et *perlxtut* pour lier C et Perl ensemble. Il peut y en avoir plus au moment où vous lisez ces lignes. Les URL suivants peuvent aussi vous servir :

<http://perldoc.perl.org/>

<http://bookmarks.cpan.org/search.cgi?cat=Training%2FTutorials>

### 19.1.9 Quels sont les groupes de discussion concernant Perl sur Usenet ? Où puis-je poser mes questions ?

Sur Usenet, plusieurs groupes sont consacrés à Perl :

|                                                 |                                                                                       |
|-------------------------------------------------|---------------------------------------------------------------------------------------|
| <code>comp.lang.perl.announce</code>            | Moderated announcement group<br>(Groupe modéré d'annonce)                             |
| <code>comp.lang.perl.misc</code>                | High traffic general Perl discussion<br>(Groupe très actif sur Perl en général)       |
| <code>comp.lang.perl.moderated</code>           | Moderated discussion group<br>(Groupe modéré de discussion)                           |
| <code>comp.lang.perl.modules</code>             | Use and development of Perl modules<br>(Utilisation et développement de modules Perl) |
| <code>comp.lang.perl.tk</code>                  | Using Tk (and X) from Perl<br>(Utilisation de Tk (et X) depuis Perl)                  |
| <code>comp.infosystems.www.authoring.cgi</code> | Writing CGI scripts for the Web.<br>(Écriture de scripts CGI pour le Web)             |

(NDT : citons aussi pour les francophones :

`fr.comp.lang.perl` Langage de programmation Perl.)

Il y a quelques années que le groupe `comp.lang.perl` a été découpé pour former tous ces groupes et que `comp.lang.perl` a été officiellement supprimé. Bien que ce groupe existe encore sur certains serveurs, il est déconseillé de l'utiliser car vos messages n'apparaîtront pas sur les serveurs qui respectent la liste officielles des noms de groupes. Utilisez `comp.lang.perl.misc` pour les messages dont le sujet n'entre dans aucun des autres groupes plus spécifiques.

Il existe aussi une passerelle Usenet vers les mailing lists Perl sponsorisée par perl.org sur [nntp://nntp.perl.org](http://nntp.perl.org) , une interface web vers les mêmes listes sur <http://nntp.perl.org/group/> et ces listes sont aussi disponibles dans la hiérarchie `perl.*` sur <http://groups.google.com> . D'autres groupes sont aussi listés sur <http://lists.perl.org/> (accessible aussi par <http://lists.cpan.org/> ).

Un endroit sympa pour poser des questions est le site PerlMonks, <http://www.perlmonks.org/> , ou la mailing liste des débutants Perl <http://lists.perl.org/showlist.cgi?name=beginners> . Pour les francophones, le forum `fr.comp.lang.perl` est aussi une bonne ressource.

Notez que personne n'est supposé écrire du code pour vous dans les ressources citées ci-dessus : il est évidemment possible de demander des conseils généraux ou des solutions pour des problèmes précis, mais n'espérez pas que quelqu'un développera du code gratuitement pour vous.

### 19.1.10 Où puis-je poster mon code source ?

Vous devriez poster votre code source dans le groupes de discussion le plus approprié mais vous pouvez aussi le cross-poster vers comp.lang.perl.misc. Si vous voulez cross-poster vers alt.sources, vérifiez que vous respectez les règles de ce groupe... En particulier l'en-tête Followup-To ne doit PAS contenir alt.sources; voir leur FAQ (<http://www.faqs.org/faqs/alt-sources-intro/>) pour plus de détails.

Si vous cherchez des logiciels, utilisez tout d'abord Google ( <http://www.google.com> ), ou l'interface de recherche sur Usenet de Google ( <http://groups.google.com> ), ou la recherche CPAN ( <http://search.cpan.org> ). C'est plus rapide et productif que de demander.

### 19.1.11 Les livres sur Perl

Il existe de nombreux livres sur Perl et/ou la programmation de CGI. Quelques-uns sont bons, d'autres corrects mais beaucoup ne valent rien ou pas grand chose. Il existe une liste de ces livres avec parfois des notes de lecture sur <http://books.perl.org/>. Si votre propre livre n'est pas listé, vous pouvez écrire à [perlfaq-workers@perl.org](mailto:perlfaq-workers@perl.org).

Le livre de référence incontestable sur Perl, écrit par le créateur de Perl est Programming Perl :

Programming Perl (the "Camel Book"):  
by Larry Wall, Tom Christiansen, and Jon Orwant  
ISBN 0-596-00027-8 [3rd edition July 2000]  
<http://www.oreilly.com/catalog/ppperl3/>  
(en Anglais, nombreuses traductions disponibles)

en français :

Programmation en Perl (le "Camel Book")  
par Larry Wall, Tom Christiansen, Jon Orwant  
3e édition, décembre 2001  
ISBN : 2-84177-140-7

Le compagnon idéal du "Camel Book" contient des milliers d'exemples réels, de mini-guides et de programmes complets :

The Perl Cookbook (the "Ram Book"):  
par Tom Christiansen et Nathan Torkington,  
avant-propos de Larry Wall  
ISBN 0-596-00313-7 [2nd Edition August 2003]  
<http://www.oreilly.com/catalog/perlckbk2/>

Si vous êtes déjà un programmeur endurci alors le "Camel Book" devrait vous suffire pour apprendre Perl. Sinon, regardez le "Llama Book" :

Learning Perl  
by Randal L. Schwartz, Tom Phoenix, and brian d foy  
ISBN 0-596-10105-8 [4th edition July 2005]  
<http://www.oreilly.com/catalog/learnperl4/>

en français :

Introduction à Perl  
Randal L. Schwartz, Tom Phoenix  
4e édition, mars 2006  
ISBN : 2-84177-404-X

Et pour des informations plus pointues sur le développement de gros programmes, présentées dans un style similaire à celui du Llama Book, vous pouvez continuer votre formation par le Alpaca Book :

Learning Perl Objects, References, and Modules (the "Alpaca Book")  
by Randal L. Schwartz, with Tom Phoenix (foreword by Damian Conway)  
ISBN 0-596-00478-8 [1st edition June 2003]  
<http://www.oreilly.com/catalog/lrnperlorm/>

Si vous n'êtes pas un programmeur occasionnel mais plutôt un programmeur confirmé voir même un informaticien de haut niveau qui n'a pas besoin d'un accompagnement pas à pas comme celui que nous proposons dans le "Llama Book", vous pouvez regarder du côté du merveilleux ouvrage :

Perl: The Programmer's Companion  
 by Nigel Chapman  
 ISBN 0-471-97563-X [1997, 3rd printing Spring 1998]  
<http://www.wiley.com/compbooks/catalog/97563-X.htm>  
<http://www.wiley.com/compbooks/chapman/perl/perltpc.html> (errata etc)

Si vous êtes un familier de Windows, l'ouvrage suivant, bien qu'un peu ancien, vous intéressera :

Learning Perl on Win32 Systems (the "Gecko Book")  
 by Randal L. Schwartz, Erik Olson, and Tom Christiansen,  
 with foreword by Larry Wall  
 ISBN 1-56592-324-3 [1st edition August 1997]  
<http://www.oreilly.com/catalog/lperlwin/>

Addison-Wesley ( <http://www.awlonline.com/> ) et Manning ( <http://www.manning.com/> ) publient eux aussi quelques bons livres sur Perl tels <Object Oriented Programming with Perl> par Damian Conway et *Network Programming with Perl* par Lincoln Stein.

Un excellent distributeur à bas prix est Bookpool sur <http://www.bookpool.com/> aevc couramment des remises de 30% ou plus.

Ce qui suit est une liste de livres que les auteurs de la FAQ ont trouvés personnellement utiles. Votre avis peut être différent. Les livres recommandés portant sur (ou principalement sur) Perl.

#### References (Références)

Programming Perl  
 by Larry Wall, Tom Christiansen, and Jon Orwant  
 ISBN 0-596-00027-8 [3rd edition July 2000]  
<http://www.oreilly.com/catalog/ppperl3/>

Perl 5 Pocket Reference  
 by Johan Vromans  
 ISBN 0-596-00032-4 [3rd edition May 2000]  
<http://www.oreilly.com/catalog/perlpr3/>

#### Tutorials (Guides d'apprentissage)

Beginning Perl  
 by James Lee  
 ISBN 1-59059-391-X [2nd edition August 2004]  
<http://apress.com/book/bookDisplay.html?bID=344>

Elements of Programming with Perl  
 by Andrew L. Johnson  
 ISBN 1-884777-80-5 [1st edition October 1999]  
<http://www.manning.com/Johnson/>

Learning Perl  
 by Randal L. Schwartz, Tom Phoenix, and brian d foy  
 ISBN 0-596-10105-8 [4th edition July 2005]  
<http://www.oreilly.com/catalog/learnperl4/>

Learning Perl Objects, References, and Modules  
 by Randal L. Schwartz, with Tom Phoenix (foreword by Damian Conway)  
 ISBN 0-596-00478-8 [1st edition June 2003]  
<http://www.oreilly.com/catalog/lrnperlorm/>

#### Task-Oriented (Orientations plus pratique)

Writing Perl Modules for CPAN  
 by Sam Tregar  
 ISBN 1-59059-018-X [1st edition Aug 2002]  
<http://apress.com/book/bookDisplay.html?bID=14>

The Perl Cookbook  
 by Tom Christiansen and Nathan Torkington  
 with foreword by Larry Wall  
 ISBN 1-56592-243-3 [1st edition August 1998]  
<http://www.oreilly.com/catalog/cookbook/>

Effective Perl Programming  
 by Joseph Hall  
 ISBN 0-201-41975-0 [1st edition 1998]  
<http://www.awl.com/>

Real World SQL Server Administration with Perl  
 by Linchi Shea  
 ISBN 1-59059-097-X [1st edition July 2003]  
<http://apress.com/book/bookDisplay.html?bID=171>

### Special Topics (Sujets spéciaux)

Perl Best Practices  
 by Damian Conway  
 ISBN: 0-596-00173-8 [1st edition July 2005]  
<http://www.oreilly.com/catalog/perlbp/>

Higher Order Perl  
 by Mark-Jason Dominus  
 ISBN: 1558607013 [1st edition March 2005]  
<http://hop.perl.plover.com/>

Perl 6 Now: The Core Ideas Illustrated with Perl 5  
 by Scott Walters  
 ISBN 1-59059-395-2 [1st edition December 2004]  
<http://apress.com/book/bookDisplay.html?bID=355>

Mastering Regular Expressions  
 by Jeffrey E. F. Friedl  
 ISBN 0-596-00289-0 [2nd edition July 2002]  
<http://www.oreilly.com/catalog/regex2/>

Network Programming with Perl  
 by Lincoln Stein  
 ISBN 0-201-61571-1 [1st edition 2001]  
<http://www.awlonline.com/>

Object Oriented Perl  
 Damian Conway  
 with foreword by Randal L. Schwartz  
 ISBN 1-884777-79-1 [1st edition August 1999]  
<http://www.manning.com/Conway/>

Data Munging with Perl  
 Dave Cross  
 ISBN 1-930110-00-6 [1st edition 2001]  
<http://www.manning.com/cross>

Mastering Perl/Tk  
 by Steve Lidie and Nancy Walsh  
 ISBN 1-56592-716-8 [1st edition January 2002]  
<http://www.oreilly.com/catalog/mastperltk/>

Extending and Embedding Perl  
 by Tim Jenness and Simon Cozens  
 ISBN 1-930110-82-0 [1st edition August 2002]  
<http://www.manning.com/jenness>

Perl Debugger Pocket Reference  
 by Richard Foley  
 ISBN 0-596-00503-2 [1st edition January 2004]  
<http://www.oreilly.com/catalog/perldebugpr/>

(NDT: un certain nombre de ces ouvrages existent sûrement en français. Si vous en connaissez les références exactes, n'hésitez pas à les envoyer au traducteur de cette FAQ.)

### 19.1.12 Quels sont les revues ou magazines parlant de Perl

Le premier (et pendant longtemps, le seul) périodique dédié à Perl est *The Perl Journal* (TPJ). Il propose des guides, des démonstrations, des études de cas, des annonces, des concours et plein d'autres choses encore. Des rubriques du TPJ concernent le développement WEB, les bases de données, Perl sur Win32, la programmation graphique, les expressions régulières et les réseaux. Le TPJ sponsorise le Obfuscated Perl Contest (le concours de Perl Obscur) et le Perl Poetry Contests (le concours de Poème en Perl). Depuis novembre 2002, le TPJ est devenu un e-zine mensuel financé par ses lecteurs qui, une fois abonnés, récupèrent chaque numéro en PDF. Pour en savoir plus, voir <http://www.tpj.com/>.

À part cela, d'autres magazines proposent souvent des articles de haut niveau sur Perl. Parmi eux: *The Perl Review* ( <http://www.theperlreview.com> ), *Unix Review* ( <http://www.unixreview.com/> ), *Linux Magazine* ( <http://www.linuxmagazine.com/> ) et le journal/lettre d'information de Usenix à ses membres `::login:` ( <http://www.usenix.org/> ).

Les articles Perl de Randal L. Schwartz sont disponibles sur le web sur <http://www.stonehenge.com/merlyn/WebTechniques/> , <http://www.stonehenge.com/merlyn/UnixReview/> et <http://www.stonehenge.com/merlyn/LinuxMag/> .

### 19.1.13 Quelles sont les listes de diffusion concernant Perl ?

La plupart des modules importants (Tk, CGI, libwww-perl) ont leur propre liste de diffusion. Lisez la documentation accompagnant ces modules pour connaître la méthode d'inscription.

Une liste exhaustive des liste de diffusion concernant Perl est consultable sur :

<http://lists.perl.org/>

### 19.1.14 Où trouver les archives de comp.lang.perl.misc ?

Le moteur de recherche de Google donne maintenant accès aux archives et aux contenus des newsgroups.

<http://groups.google.com/groups?group=comp.lang.perl.misc>

ou pour les francophones

<http://groups.google.com/groups?group=fr.comp.lang.perl>

Si vous avez une question, vous pouvez être sûr que quelqu'un d'autre l'a déjà posée sur comp.lang.perl.misc (ou pour les francophones fr.comp.lang.perl). Avec un peu de temps et de patience pour passer au crible toutes les discussions, vous avez de grandes chances de trouver la réponse qu'il vous faut.

### 19.1.15 Où puis-je acheter une version commerciale de Perl ?

D'un certain point de vue réaliste, Perl *est* déjà un logiciel commercial : il possède une licence que vous pouvez récupérer et faire lire à votre chef. Il est utilisé par de très nombreuses personnes et il existe énormément de livres le concernant. Il est distribué sous forme de versions avec des packages clairement définis. Les groupes de discussion compl.lang.perl.\*, fr.comp.lang.perl et plusieurs mailing lists fournissent gratuitement des réponses à vos questions et ce quasiment en temps réel. La maintenance de Perl est traditionnellement assuré par Larry, un bon nombre de développeurs et des myriades de programmeurs. Tous travaillent gratuitement pour créer quelque chose permettant de rendre la vie plus facile à tous.

Cependant, cette réponse peut ne pas suffire aux gestionnaires qui réclament un bon de livraison provenant d'une entreprise qu'ils pourraient poursuivre en justice en cas de problème. Ou peut-être ont-ils besoin d'un vrai support avec des obligations contractuelles. Il existe des CD commerciaux contenant perl si cela peut aider. Beaucoup de livres sur Perl contiennent par exemple une distribution de Perl, comme les Perl Resource Kits d'O'Reilly (à la fois en version Unix et en version propriétaire Microsoft) ; les distributions libres d'Unix sont toutes livrées avec Perl.

### 19.1.16 Où dois-je envoyer mes rapports de bugs ?

Si vous avez trouvé un bug dans l'interpréteur Perl ou dans les modules fournis avec Perl, utilisez le programme *perlbug* fourni dans la distribution ou envoyez votre rapport à [perlbug@perl.com](mailto:perlbug@perl.com) ou sur <http://rt.perl.org/perlbug/> .

Pour les modules Perl, vous pouvez soumettre votre rapport au traqueur sur <http://rt.cpan.org> .

Si votre bug concerne un portage non-standard (voir la question Quelles machines supportent perl ? Où puis-je trouver Perl ? (§19.1.1)), une distribution binaire ou un module non-standard (tel que Tk, CGI, etc.) alors veuillez consultez leur propre documentation pour déterminer le moyen exact d'envoyer votre rapport.

Lisez *perlbug* (perl5.004 ou plus) plus plus de détails.



### 19.1.17 Qu'est-ce que perl.com ? Les Perl Mongers ? pm.org ? perl.org ? cpan.org ?

Perl.com sur <http://www.perl.com/> fait partie de O'Reilly Network, une filiale de O'Reilly Media.

La Fondation Perl est une organisation de soutien du langage Perl qui maintient le site web <http://www.perl.org/> pour promouvoir le langage Perl. Ce domaine est utilisé pour fournir des services d'aide à la communauté Perl incluant l'hébergement de listes de diffusion, de sites web et d'autres services. Le site web <http://www.perl.org/> lui-même est le site général de promotion du langage Perl et il existe plusieurs sous-domaines pour des besoins spécifiques tels que :

```
http://learn.perl.org/
http://use.perl.org/
http://jobs.perl.org/
http://lists.perl.org/
```

Les Perl Mongers utilisent le domaine pm.org pour des services liés aux groupes d'utilisateurs de Perl incluant l'hébergement de listes de diffusion, de sites web. Pour savoir comment rejoindre, créer ou faire appel aux services d'un groupe d'utilisateurs Perl, consultez le site web <http://www.pm.org/>.

<http://www.cpan.org/> est le Comprehensive Perl Archive Network, un dépôt de programme Perl dupliqué mondialement. Voir la question *What is CPAN?* plus haut dans ce document. =head1 AUTEUR ET COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen et Nathan Torkington et autres auteurs sus-mentionnés. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code en précisant l'origine serait de bonne courtoisie mais n'est pas obligatoire.

## 19.2 TRADUCTION

La traduction française est distribuée avec les mêmes droits que sa version originale (voir ci-dessus).

### 19.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 19.2.2 Traducteur

Traduction et mise à jour Paul Gaborit <Paul.Gaborit@enstimac.fr>.

### 19.2.3 Relecture

Pascal Ethvignot <pascal@encelade.frmug.org>, Roland Trique <roland.trique@uhb.fr>.

# Chapitre 20

## perlfaq3

Outils de programmation

### 20.1 DESCRIPTION

Cette section de la FAQ répond à des questions relatives aux outils de programmation et à l'aide de programmation.

#### 20.1.1 Comment fais-je pour... ?

Avez-vous déjà été voir le CPAN (voir *perlfaq2*) ? Il y a des chances pour que quelqu'un ait déjà écrit un module susceptible de résoudre votre problème. Avez-vous déjà lu les pages man appropriées ? Voilà un bref sommaire :

|                      |                                                                                                                                                           |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bases                | perldata, perlvar, perlsyn, perlop, perlsub                                                                                                               |
| Exécution            | perlrun, perldebug                                                                                                                                        |
| Fonctions            | perlfunc                                                                                                                                                  |
| Objets               | perlref, perlmod, perlobj, perltie                                                                                                                        |
| Structure de données | perlref, perllool, perldsc                                                                                                                                |
| Modules              | perlmod, perlmodlib, perlsub                                                                                                                              |
| Regexp               | perlre, perlfunc, perlop, perllocale                                                                                                                      |
| Évoluer vers perl5   | perltrap, perl                                                                                                                                            |
| Lier au langage C    | perlxs, perlcall, perlguts, perlembed                                                                                                                     |
| Divers               | <a href="http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz">http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz</a><br>(ce n'est pas une page man, mais très utile) |

Un sommaire rudimentaire des pages de manuel de Perl existantes se trouve dans *perltoc*.

#### 20.1.2 Comment utiliser Perl de façon interactive ?

L'approche typique consiste à utiliser le débogueur Perl, décrit dans *perldebug*, avec un programme "vide", comme cela :

```
perl -de 42
```

Maintenant, tapez plutôt un code Perl valide, et il sera immédiatement évalué (exécuté). Vous pouvez également examiner la table des symboles, voir l'évolution de la pile, vérifier les valeurs des variables, fixer des points d'arrêt, et faire d'autres opérations typiquement disponibles dans les débogueur symboliques.

#### 20.1.3 Existe-t-il un shell Perl ?

Le psh (Perl sh) en est à sa version 1.8. C'est un shell qui combine la nature interactive de shell Unix avec la puissance de Perl. Son but est d'être un shell complet utilisant la syntaxe et les fonctionnalités de Perl pour les structures de contrôles et d'autres choses. Vous pouvez récupérer psh sur <http://sourceforge.net/projects/psh/>.

Zoidberg est un projet similaire qui fournit un shell développé en perl, configuré en perl et utilisé en perl. Il peut être utilisé comme shell de login et comme shell de développement. Vous pouvez le trouver sur <http://zoidberg.sf.net> ou sur votre miroir CPAN.

Le module Shell.pm (distribué avec Perl) fait que Perl tente l'exécution des commandes qui ne font pas partie du langage Perl en tant que commandes shell. perlsh de la distribution source est simpliste et inintéressant, mais correspondra peut-être à ce que vous recherchez.

### 20.1.4 Comment puis-je connaître les modules installés sur mon système ?

Vous pouvez utiliser le module `ExtUtils::Installed` pour voir toutes les distributions installées. Il peut prendre un peu de temps pour produire sa magie. Les bibliothèques standards qui viennent avec Perl sont regroupées sous le nom "Perl" (vous pouvez les voir via `Module::CoreList`).

```
use ExtUtils::Installed;

my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

Si vous souhaitez une liste de tous les noms de fichiers des modules Perl, vous pouvez utiliser `File::Find::Rule`.

```
use File::Find::Rule;

my @files = File::Find::Rule->file()->name('*.pm')->in(@INC);
```

Si vous ne disposez pas de ce module, vous pouvez obtenir le même résultat via `File::Find` qui fait partie de la distribution standard.

```
use File::Find;
my @files;
find(
 sub {
 push @files, $File::Find::name
 if -f $File::Find::name && /\.pm$/
 },

 @INC
);

print join "\n", @files;
```

Si vous voulez juste vérifier rapidement qu'un module est disponible, vous pouvez chercher sa documentation. Si vous pouvez lire la documentation d'un module, il y a de grandes chances qu'il soit installé. Si vous ne le pouvez pas, c'est peut-être que le module n'en propose pas (dans de très rares cas).

```
prompt% perldoc Nom::Module
```

Vous pouvez aussi essayer d'inclure ce module dans un script en une ligne pour voir si perl le trouve.

```
prompt% perldoc -MNom::Module -e1
```

### 20.1.5 Comment déboguer mes programmes Perl ?

Avez-vous essayé `use warnings` ou utilisé `-w` ? Cela permet d'afficher des avertissements (des warnings) pour détecter les pratiques douteuses.

Avez-vous utilisé `use strict` ? Cela vous empêche d'utiliser des références symboliques, vous oblige à prédéclarer les sous-programmes que vous appelez comme simple mot, et (probablement le plus important) vous oblige à déclarer vos variables avec `my`, `our` ou `use vars`.

Avez-vous vérifié les résultats de chacune des commandes système ? Le système d'exploitation (et donc Perl) vous indique si elles ont fonctionné ou pas, et la raison de l'échec éventuel.

```
open(FH, "> /etc/cantwrite")
 or die "Couldn't write to /etc/cantwrite: $!\n";
```

Avez-vous lu *perltrap* ? C'est plein de trucs et astuces pour les programmeurs Perl débutants ou initiés. Il y a même des sections pour ceux d'entre vous qui viennent des langages *awk* et *C*.

Avez-vous essayé le débogueur Perl, décrit dans *perldebug* ? Vous pouvez exécuter votre programme et voir ce qu'il fait, pas à pas et ainsi comprendre pourquoi ce qu'il fait n'est pas conforme à ce qu'il devrait faire.

### 20.1.6 Comment mesurer les performances de mes programmes Perl ?

Vous devriez utiliser le module `Devel::DProf` dans les distributions standard récentes (ou de CPAN), ainsi que `Benchmark.pm` de la distribution standard. `Benchmark` vous permet de mesurer le temps d'exécution de portions spécifiques de votre code alors que `Devel::DProf` vous donne des détails sur les endroits où le code consomme du temps.

Voici un exemple d'utilisation de `Benchmark` :

```
use Benchmark;

@junk = 'cat /etc/motd';
$count = 10_000;

timethese($count, {
 'map' => sub { my @a = @junk;
 map { s/a/b/ } @a;
 return @a },
 'for' => sub { my @a = @junk;
 local $_;
 for (@a) { s/a/b/ };
 return @a },
});
```

Voici l'affichage généré (sur une machine particulière—vos résultats dépendront de votre matériel, du système d'exploitation, et de la charge de travail de votre machine) :

```
Benchmark: timing 10000 iterations of for, map...
 for: 4 secs (3.97 usr 0.01 sys = 3.98 cpu)
 map: 6 secs (4.97 usr 0.00 sys = 4.97 cpu)
```

Soyez conscient qu'un bon benchmark est très difficile à écrire. Il ne teste que les données que vous lui passez, et ne prouve vraiment que peu de choses sur les différences de complexités entre divers algorithmes.

### 20.1.7 Comment faire une liste croisée des appels de mon programme Perl ?

Le module `B::Xref` peut être utilisé pour générer une liste croisée des appels pour les programmes Perl.

```
perl -MO=Xref[,OPTIONS] scriptname.plx
```

### 20.1.8 Existe-t-il un outil de mise en page de code Perl ?

`Perltidy` est un script Perl qui indente et reformate les scripts Perl pour les rendre plus lisibles en essayant de respecter les règles de *perlstyle*. Si vous écrivez des scripts Perl ou passez beaucoup de temps à en lire, vous l'apprécierez sûrement. Il est disponible sur <http://perltidy.sourceforge.net>.

Bien sûr, si vous respectez les recommandations de *perlstyle*, vous ne devriez pas avoir besoin de reformater. L'habitude de formater votre code au fur et à mesure que vous l'écrivez vous évitera bien des erreurs. Votre éditeur devrait vous aider à le faire. Dans `emacs`, le `perl-mode` et son successeur, le `cperl-mode`, peuvent vous être d'une grande aide pour quasiment tout le code, et même d'autres éditeurs moins programmables peuvent vous fournir une assistance significative. Tom Christiansen et de nombreux utilisateurs de `vi` ne jure que par les réglages suivants sous `vi` et ses clones :

```
set ai sw=4
map! ^O {^M}^[O^T
```

Placez cela dans votre fichier `.exrc` (en remplaçant les accents circonflexes par des caractères de contrôle) et c'est bon. En mode insertion, `^T` est pour l'indentation, `^D` pour la suppression de l'indentation, et `^O` pour l'indentation d'un bloc. Un exemple plus complet, avec des commentaires, peut être trouvé sur <http://www.cpan.org/authors/id/TOMC/scripts/toms.exrc.gz>.

Le programme `a2ps` sur <http://www-inf.enst.fr/%7Edemaille/a2ps/black+white.ps.gz> fait beaucoup pour imprimer des sorties de documents joliment imprimées. Voyez aussi `enscript` sur <http://people.ssh.fi/mtr/genscript/>.

### 20.1.9 Existe-t-il un ctags pour Perl ?

(contribution de brain d foy)

Exuberent ctags reconnaît Perl : <http://ctags.sourceforge.net/>.

Vous pouvez aussi essayer pltags : <http://www.mscha.com/pltags.zip>.

### 20.1.10 Existe-t-il un environnement de développement intégré (IDE) ou un éditeur Perl sous Windows ?

Les programmes Perl sont juste du texte donc n'importe quel éditeur peut convenir.

Si vous êtes sur Unix, vous disposez déjà d'un IDE : Unix lui-même. La philosophie Unix se traduit par un ensemble de petits outils qui réalisent chacun une seule tâche mais bien. C'est comme une boîte à outils.

Si vous souhaitez absolument un IDE, consultez la liste suivante (par ordre alphabétique et non par ordre de préférence) :

#### Eclipse

<http://e-p-i-c.sf.net/>

Le projet d'intégration de Perl dans Eclipse propose l'edition et le débogage avec Eclipse.

#### Enginsite

<http://www.enginsite.com/>

Perl Editor par EngInSite est un environnement de développement complètement intégré (IDE) pour créer, tester et déboguer des scripts Perl. Il tourne sur Windows 9x/NT/2000/XP ou plus.

#### Komodo

<http://www.ActiveState.com/Products/Komodo/>

L'IDE d'ActiveState multi-plateformes (en octobre 2004, cela concernait Windows, Linux et Solaris), multi-langage IDE connaît Perl et inclut un débogueur d'expressions rationnelles et du débogage distant.

#### Open Perl IDE

<http://open-perl-ide.sourceforge.net/>

Open Perl IDE est un environnement de développement intégré permettant l'écriture et le débogage de scripts Perl avec la distribution ActivePerl d'ActiveState sous Windows 95/98/NT/2000.

#### OptiPerl

<http://www.optiperl.com/>

OptiPerl est un IDE Windows permettant la simulation de l'environnement CGI et incluant un débogueur et un éditeur avec décoration syntaxique.

#### PerlBuilder

<http://www.solutionsoft.com/perl.htm>

PerlBuidler est un environnement de développement intégré pour Windows qui permet les développement Perl.

#### visiPerl+

<http://helpconsulting.net/visiperl/>

De Help Consulting, pour Windows.

#### Visual Perl

[http://www.activestate.com/Products/Visual\\_Perl/](http://www.activestate.com/Products/Visual_Perl/)

Visual Perl est le pug-in Visual Studio.NET d'ActiveState.

#### Zeus

<http://www.zeusedit.com/lookmain.html>

Zeus pour Window est un autre éditeur/IDE multi-langages pour Win32 qui vient avec le support Perl.

Pour les éditeurs : si vous êtes sous Unix vous disposez certainement de vi ou d'un de ses clones et peut-être aussi d'emacs. Vous n'avez donc rien à télécharger. Dans emacs, le mode cperl-mode (M-x cperl-mode) vous fournira un mode d'édition Perl parmi les meilleurs de tous les éditeurs.

Si vous utilisez Windows, vous pouvez utiliser n'importe quel éditeur qui permet d'éditer du texte brut, tel que NotePad, WordPad ou le Bloc-Notes. Les traitements de texte comme Microsoft Word ou WordPerfect ne fonctionnent pas bien car ils insèrent plein d'informations cachées (ceci étant, la plupart proposent aussi l'enregistrement au format "Texte seul"). Vous pouvez aussi télécharger des éditeurs de textes conçus spécialement pour la programmation comme Textpad (<http://www.textpad.com>) et UltraEdit (<http://www.ultraedit.com>) entre autres.

Si vous utilisez MacOS, les mêmes conseils s'appliquent. MacPerl (pour l'environnement Classic) vient avec un éditeur simple. Les éditeurs externes les plus populaires sont BEdit (<http://www.bbedit.com/>) or Alpha (<http://www.his.com/~jguyer/Alpha/Alpha8.html>). Les utilisateurs de MacOS X peuvent aussi utiliser les éditeurs Unix. Neil Bowers (l'homme derrière Geekcruises) propose une liste des éditeurs connaissant Perl sur Mac (<http://www.neilbowers.org/macperleditors.html>).

**GNU Emacs**

<http://www.gnu.org/software/emacs/windows/ntemacs.html>

**MicroEMACS**

<http://www.microemacs.de/>

**XEmacs**

<http://www.xemacs.org/Download/index.html>

**Jed**

<http://space.mit.edu/~davis/jed/>

ou les clones de vi tels :

**Elvis**

<ftp://ftp.cs.pdx.edu/pub/elvis/> <http://www.fh-wedel.de/elvis/>

**Vile**

<http://dickey.his.com/vile/vile.html>

**Vim**

<http://www.vim.org/>

Pour les amoureux de vi en général, sous Windows ou autres :

<http://www.thomer.com/thomer/vi/vi.html>

nvi (<http://www.bostic.com/vi/>, disponible sur CPAN dans src/misc) est encore un autre clone de vi, malheureusement sans version Windows, qui peut être intéressant sur les plateformes Unix car, d'une part, ce n'est pas à proprement parler un clone de vi mais bien le vrai vi ou plutôt son évolution et, d'autre part, parce qu'il peut utiliser Perl comme langage de programmation interne. nvi n'est pas le seul dans ce cas : vim et vile offrent cette possibilité de Perl intégré.

Les programmes suivants sont des éditeurs Win32 multi-langages qui supportent Perl :

**Codewright**

<http://www.borland.com/codewright/>

**MultiEdit**

<http://www.MultiEdit.com/>

**SlickEdit**

<http://www.slickedit.com/>

Il existe aussi un petit éditeur de texte gadget écrit en Perl et qui est distribué dans le module Tk sur CPAN. ptkdb (<http://world.std.com/~aep/ptkdb/>) est débogueur Perl/Tk qui agit un peu comme une sorte d'environnement de développement. Perl Composer (<http://perlcomposer.sourceforge.net/>) est un IDE pour la création d'IHM (Interface Homme Machine) en Perl/Tk.

En plus d'un éditeur/IDE, vous aurez sans doute besoin d'un shell plus puissant en environnement Win32. Voici quelques possibilités :

**Bash**

fourni comme paquetage Cygwin (<http://sources.redhat.com/cygwin/>).

**Ksh**

depuis la boîte à outils MKS (<http://www.mks.com/>), ou le Bourne shell de l'environnement U/WIN (<http://www.research.att.com/sw/tools/uwin/>)

**Tcsh**

<ftp://ftp.astron.com/pub/tcsh/>, voir aussi <http://www.primate.wisc.edu/software/csh-tcsh-book/>

**Zsh**

<ftp://ftp.blarg.net/users/amol/zsh/>, voir aussi <http://www.zsh.org/>

MKS et U/WIN sont commerciaux (U/WIN est gratuit dans un cadre éducation ou recherche) alors que Cygwin repose sur la licence Publique GNU (mais ça ne compte pas pour une utilisation avec Perl). Cygwin, MKS et U/WIN proposent chacun, en plus de leurs shells respectifs, un ensemble complet d'outils standard Unix.

Si vous transférez des fichiers textes entre Unix et Windows en utilisant FTP, assurez-vous de les faire en mode ASCII afin que les caractères de fin de ligne soient correctement convertis.

Sur MacOS, l'application MacPerl propose un éditeur de texte simple limité à 32k qui peut se comporter comme un IDE rudimentaire. À l'opposé, l'outil MPW Perl peut utiliser le shell MPW lui-même comme un éditeur (sans la limite des 32k).

### Affrus

est un véritable environnement de développement en Perl avec un débogueur complet (<http://www.latenightsw.com>).

### Alpha

est un éditeur écrit et extensible en Tcl qui supporte de nombreux langages de programmation ou à base de balises tels que Perl et HTML (<http://www.his.com/~jguyer/Alpha/Alpha8.html>).

### BBEdit et BBEdit Lite

sont des éditeurs de texte pour Mac OS qui ont un mode reconnaissant le Perl (<http://web.barebones.com/>).

Pepper et Pe sont des éditeurs de texte gérant les langages de programmation et développés respectivement pour Mac OS X et BeOS (<http://www.hekkelman.com/>).

## 20.1.11 Où puis-je trouver des macros pour Perl sous vi ?

Pour une version complète du fichier de configuration de Tom Christiansen pour vi, voyez [http://www.cpan.org/authors/Tom\\_Christiansen/scripts/toms.exrc.gz](http://www.cpan.org/authors/Tom_Christiansen/scripts/toms.exrc.gz), si s'agit du fichier standard pour les émulateurs vi. Cela fonctionne mieux avec nvi, la dernière version de vi de Berkeley, qui peut éventuellement être compilée avec un interpréteur Perl inclus – voir <http://www.cpan.org/src/misc/>.

## 20.1.12 Où puis-je trouver le mode perl pour emacs ?

Depuis la version 19 patchlevel 22 de Emacs, perl-mode.el et l'aide pour le débogueur Perl sont inclus. Vous devriez l'avoir dans la distribution standard version 19 d'Emacs.

Dans le répertoire du code source de Perl, vous trouverez un répertoire nommé "emacs", qui contient un mode d'édition perl qui colore les instructions Perl, fournit une aide contextuelle, et autres choses sympathiques.

Notez que le mode perl de Emacs changera les lignes du genre "main' foo" (apostrophe), et modifiera les tabulations et surlignages. Vous utilisez probablement de toute façon "main::foo" dans votre nouveau code Perl, donc ceci n'est pas bien grave.

## 20.1.13 Comment utiliser des 'curses' avec Perl ?

Le module Curses du CPAN fournit une interface objet chargeable dynamiquement pour la librairie "curses". Une petite démo se trouve dans le répertoire [http://www.cpan.org/authors/Tom\\_Christiansen/scripts/rep.gz](http://www.cpan.org/authors/Tom_Christiansen/scripts/rep.gz); Ce programme répète régulièrement une commande et rafraîchit l'écran comme il faut, rendant ainsi **rep ps axu** similaire à **top**.

## 20.1.14 Comment puis-je utiliser X ou Tk avec Perl ?

Les modules Tk forment une interface pour les outils Tk, entièrement orientée objet et basée sur Perl qui vous évite d'utiliser Tcl pour avoir accès à Tk. Sx est une interface pour l'ensemble Athena Widget. Les deux sont disponibles sur CPAN. Voyez le répertoire [http://www.cpan.org/modules/by-category/08\\_User\\_Interfaces/](http://www.cpan.org/modules/by-category/08_User_Interfaces/).

D'incalculables aides pour la programmation en Perl/Tk sont la FAQ Perl/Tk sur <http://phaseit.net/claird/comp.lang.perl.tk/ptkFAQ.html>, le guide de référence Perl/Tk disponible sur [http://www.cpan.org/authors/Stephen\\_O\\_Lidie/](http://www.cpan.org/authors/Stephen_O_Lidie/) et les pages de manuel en ligne sur <http://www-users.cs.umn.edu/%7Eamundson/perl/perltk/toc.html>.

### 20.1.15 Comment rendre mes programmes Perl plus rapides ?

La meilleure façon pour y parvenir est d'utiliser un meilleur algorithme. Cela peut souvent faire une énorme différence. Le livre de Jon Bentley *Programming Pearls* (sans faute de frappe !) donne aussi de bonnes astuces d'optimisation. Quelques conseils pour l'amélioration de la vitesse : utilisez benchmark pour être certain que vous optimisez la bonne partie de votre code, cherchez de meilleurs algorithmes plutôt que des micro-optimisations pour votre code actuel, et si rien ne s'arrange, considérez qu'il vous faut juste acheter un matériel plus rapide. Si vous ne l'avez pas déjà fait, vous deviez probablement lire la réponse à la question posée plus haut "Comment mesurer les performances de mes programmes Perl ?".

Une approche différente est d'utiliser Autoload pour le code peu utilisé. Voyez les modules AutoSplit et AutoLoader dans la distribution standard pour ce faire. Vous pouvez aussi localiser le goulot d'étranglement et penser à écrire cette partie en C, de la même façon que nous avons l'habitude d'écrire les parties lentes de C en assembleur. Au lieu de réécrire en C, vous pouvez aussi utiliser des modules ayant leurs sections critiques déjà écrites en C (par exemple, le module PDL du CPAN).

Si vous liez votre exécutable perl à une librairie partagée *libc.so*, vous pouvez souvent gagner entre 10 % et 25 % en performance en le liant plutôt avec une librairie statique *libc.a*. Cela fera un exécutable plus important, mais vos programmes (et programmeurs) Perl vous en remercieront. Voyez le fichier *INSTALL* dans le code source de la distribution pour plus d'informations.

Le programme undump était une ancienne tentative visant à améliorer la vitesse des programmes Perl en les sauvegardant sous forme déjà compilée sur le disque. Ce n'est plus une option viable, puisque cela ne fonctionnait que sur peu d'architectures, et que ce n'était pas une bonne solution de toute façon.

### 20.1.16 Comment faire pour que mes programmes Perl occupent moins de mémoire ?

Quand Perl a des échanges internes trop longs, il préfère imputer cela à un problème de mémoire. Les scalaires en Perl utilisent plus de mémoire que les chaînes de caractères en C, les tableaux en utilisent plus aussi, et les tables de hachage moins. Bien qu'il y ait encore beaucoup à faire, les versions récentes répondent à ces problèmes. Par exemple, dans la version 5.004, les clés dupliquées des tables de hachage sont partagées par toutes les tables de hachage les utilisant, ne nécessitant pas de ré-allocation.

Dans certains cas, l'usage de `substr()` ou `vec()` pour simuler des tableaux peut être très bénéfique. Par exemple, un tableau d'un millier de booléens prendra au moins 20 000 octets, mais il peut être remplacé par un vecteur de 125 octets – réalisant ainsi une économie considérable de mémoire. Le module standard `Tie::SubstrHash` peut aussi aider pour certains types de structures de données. Si vous travaillez avec des structures de données spécifiques (matrices, par exemple), les modules qui les implémentent en C peuvent utiliser moins de mémoire que leurs équivalents en Perl.

Autre chose, essayez de savoir si Perl a été compilé avec le `malloc` système ou le `malloc` de Perl. Quel qu'il soit, essayez d'utiliser l'autre, et voyez si cela fait une différence. Les informations sur `malloc` se trouvent dans le fichier *INSTALL* du source de la distribution. Vous pouvez savoir quel `malloc` vous utilisez en tapant `perl -V:usemymalloc`.

Évidemment, le moyen le plus sûr de réduire l'occupation mémoire est d'en utiliser le moins possible dès le départ. De bonnes pratiques de programmation peuvent aider :

- Ne faites pas de slurp !

Ne lisez pas un fichier complet en mémoire si vous pouvez le traiter ligne par ligne. Plus concrètement, utilisez une boucle comme :

```
#
Bonne idée
#
while (<FILE>) {
 # ...
}
```

plutôt que :

```
#
Mauvaise idée
#
@data = <FILE>;
foreach (@data) {
 # ...
}
```

Tant que les fichiers traités sont petits, cela n'a pas beaucoup d'importance mais s'ils deviennent gros alors cela fera une énorme différence.

- Utilisez `map` et `grep` intelligemment

Souvenez-vous que `map` et `grep` utilise un argument sous forme de LISTE, donc faire :



```
@wanted = grep {/pattern/} <FILE>;
```

chargera le fichier complet en mémoire. Pour de gros fichiers, il vaut mieux utiliser une boucle :

```
while (<FILE>) {
 push(@wanted, $_) if /pattern/;
}
```

- Évitez les guillemets et les stringifications inutiles

Ne placez de grande chaînes de caractères entre guillemets si ce n'est pas absolument nécessaire. Le code suivant :

```
my $copie = "$grande_chaine";
```

réalise 2 copies de \$grande\_chaine (une pour \$copie et une autre pour les guillemets). Alors que :

```
my $copie = $grande_chaine;
```

ne fait qu'une seule copie.

C'est la même chose pour la stringification (la transformation en chaîne) de gros tableaux :

```
{
 local $, = "\n";
 print @gros_tableau;
}
```

est plus efficace d'un point de vue occupation mémoire que :

```
print join "\n", @gros_tableau;
```

ou que :

```
{
 local $" = "\n";
 print "@gros_tableau";
}
```

- Utilisez les références

Passez vos tableaux et tables de hachages par référence plutôt que par valeur. Déjà, c'est le seul moyen de passer plusieurs listes ou tables de hachage (ou les deux) en un seul appel ou return. Cela évite aussi une recopie de tout leur contenu. Par contre, il faut le faire avec précaution car tout changement sera alors propagé aux données originales. Si vous avez réellement besoin de modifier une copie, vous devrez sacrifier la mémoire nécessaire à cette copie.

- Liées les grosses variables avec le disque

Pour de très "grosses" structures de données (celles dont la taille dépasse la taille mémoire disponible), pensez à utiliser l'un des modules DB qui stocke l'information sur disque plutôt qu'en mémoire. Cela a évidemment un impact sur le temps d'accès mais ce sera très probablement plus efficace que d'utiliser de façon massive le swap mémoire sur le disque.

### 20.1.17 Est-ce sûr de retourner un pointeur sur une donnée locale ?

Oui. Le ramasse-miettes (le gestionnaire de mémoire) de Perl fait attention à cela.

```
sub makeone {
 my @a = (1 .. 10);
 return \@a;
}

for (1 .. 10) {
 push @many, makeone();
}

print $many[4][5], "\n";

print "@many\n";
```

### 20.1.18 Comment puis-je libérer un tableau ou une table de hachage pour réduire mon programme ?

(contribution de Michael Carman)

Vous ne pouvez pas. La mémoire allouée aux variables lexicales (c'est à dire les variables my()) ne peut pas être réutilisée même si on sort de sa portée (NdT: mais ce n'est heureusement pas le cas des données référencées par ces variables). Elle reste réservée au cas où la variable reviendrait à nouveau à portée. La mémoire allouée aux variables globales peut être réutilisée (par votre programme) en leur appliquant undef() ou delete().

Sur la plupart des systèmes d'exploitation, la mémoire allouée à un programme ne peut pas être retournée au système. C'est pourquoi les programmes ayant un temps d'exécution très long se ré-exécutent eux-même. Quelques systèmes d'exploitation (notamment, les systèmes utilisant `mmap(2)`) peuvent parfois récupérer certains espaces mémoires qui ne sont plus utilisés, mais, pour que cela fonctionne avec Perl, il faut que perl soit configuré et compilé pour utiliser le `malloc` du système et non celui de `perl`.

En général, vous n'avez pas à (et vous ne devriez pas) vous préoccuper de l'allocation et de la désallocation de mémoire avec Perl.

Voir aussi "Comment faire pour que mes programmes Perl occupent moins de mémoire ?".

### 20.1.19 Comment rendre mes scripts CGI plus efficaces ?

Après les mesures classiques décrites pour rendre vos programmes Perl plus rapides ou courts, il y a d'autres possibilités pour les programmes CGI. Il peut être exécuté plusieurs fois par seconde. Étant donné qu'à chaque fois, il doit être recompilé, et demande un mégaoctet ou plus de mémoire allouée, cela peut couler le système en performances. Compiler en C **ne vous avancera pas davantage** car le démarrage du processus est le point le plus lent.

Il y a 2 façons classiques pour éviter cette surcharge. Une solution consiste à exécuter le serveur HTTP Apache (disponible à <http://www.apache.org/>) avec le module `mod_perl` ou `mod_fastcgi`.

Avec `mod_perl` et le module `Apache::Registry` (distribué avec `mod_perl`), `httpd` fonctionne avec un interpréteur Perl inclus qui précompile votre script puis l'exécute dans le même espace mémoire sans `fork`. L'extension Apache donne aussi à Perl l'accès à la librairie API du serveur, ce qui fait qu'un module écrit en Perl peut faire quasiment tout ce qu'un module en C peut. Pour en savoir plus sur `mod_perl`, voyez <http://perl.apache.org/>.

Avec le module `FCGI` (du CPAN) et le module `mod_fastcgi` (disponible sur <http://www.fastcgi.com/>) chacun de vos programmes devient un processus CGI démon permanent.

Chacune de ces 2 solutions peut avoir de grandes conséquences sur votre système et sur votre façon d'écrire vos programmes CGI, donc utilisez-les avec précaution.

Voir [http://www.cpan.org/modules/by-category/15\\_World\\_Wide\\_Web\\_HTML\\_HTTP\\_CGI/](http://www.cpan.org/modules/by-category/15_World_Wide_Web_HTML_HTTP_CGI/).

### 20.1.20 Comment dissimuler le code source de mon programme Perl ?

Effacez-le. :-) Plus sérieusement, il y a bon nombre de solutions (pour la plupart peu satisfaisante) avec différents degrés de "sécurité".

Tout d'abord, en aucun cas, vous ne pouvez ôter le droit de lecture du script, car le code source doit pouvoir être lu pour pouvoir être compilé et interprété. (Ce qui ne signifie pas que le code source d'un script CGI est accessible en lecture pour les visiteurs du site web – il l'est uniquement pour ceux qui ont accès au système de fichiers). Donc vous devez laisser ces permissions au niveau socialement sympathique de 0755.

Certains voient cela comme un problème de sécurité. Si votre programme fait des actions mettant en cause la sécurité du système, et repose sur la confiance aveugle que vous avez dans le fait que les visiteurs ne savent pas comment exploiter ces trous de sécurité, alors, votre programme n'est pas sécurisé. Il est souvent possible de détecter ce genre de trous et de les exploiter sans jamais voir le code source. La sécurité qui repose sur l'obscurité, autrement dit qui repose sur la non visibilité des bugs (au lieu de les résoudre) est une sécurité très faible.

Vous pouvez essayer d'utiliser le cryptage du code via des filtres de source (depuis la version 5.8 les modules `Filter::Simple` et `Filter::Util::Call` sont inclus dans la distribution standard), mais n'importe quel programmeur sérieux pourra les décrypter. Vous pouvez essayer d'utiliser le compilateur et interpréteur en binaire décrit ci-dessous, mais les plus curieux pourront encore le décompiler. Vous pouvez essayer le compilateur en code natif décrit ci-dessous, mais des crackers peuvent encore le désassembler. Cela pose différents degrés de difficultés aux personnes qui en veulent à votre code, mais ça ne les empêchera pas de le retrouver (c'est vrai pour n'importe quel langage, pas juste pour Perl).

Il est très facile de récupérer les sources d'un programme Perl. Il vous suffit de fournir ce programme à l'interpréteur perl et d'utiliser les modules de la hiérarchie `B::*`. Le module `B::Deparse`, par exemple, fait échouer la plupart des tentatives de dissimulation de code. Encore une fois, cela n'a rien de spécifique à Perl.

Si cela vous ennuie que des personnes puissent profiter de votre code, alors la première chose à faire est d'indiquer une licence restrictive au début de votre code, ce qui vous donne une sécurité légale. Mettez une licence à votre programme et saupoudrez-le de menaces diverses telle que "Ceci est un programme privé non distribué, de la société XYZ. Votre accès à ce code source ne vous donne pas le droit de l'utiliser bla bla bla." Nous ne sommes pas juristes, bien sûr, donc vous devriez en consulter un pour vous assurer que votre texte de licence tient devant un tribunal.

### 20.1.21 Comment compiler mon programme Perl en code binaire ou C ?

(contribution de brian d foy)

De manière générale, ce n'est pas possible. Il existe dans certains cas des solutions. Les gens demandent souvent cela parce qu'ils veulent distribuer leur travail sans donner le code source. Vous ne verrez sans doute aucune amélioration des performances puisque la plupart des solutions ne font qu'intégrer un interpréteur Perl au produit final (mais voyez tout de même Comment rendre mes programmes Perl plus rapides ?).

Le Perl Archive Toolkit (<http://par.perl.org/index.cgi>) est à Perl ce que JAR est à Java. Il est disponible sur CPAN (<http://search.cpan.org/dist/PAR/>).

La hiérarchie B::\*, parfois appelée "le compilateur Perl", est en fait un moyen pour les programmes Perl d'accéder à leurs propres entrailles plutôt qu'un moyen de créer des versions pré-compilées. Mais il est vrai que le module B::Bytecode peut transformer votre script en un code binaire que vous pouvez ensuite recharger via le module ByteLoader pour l'exécuter comme un script Perl normal.

Il existe quelques produits commerciaux qui font le travail pour vous si vous achetez la licence correspondante.

Le Perl Dev Kit ([http://www.activestate.com/Products/Perl\\_Dev\\_Kit/](http://www.activestate.com/Products/Perl_Dev_Kit/)) d'ActiveState peut "transformer votre programme Perl en un fichier directement exécutable sur HP-UX, Linux, Solaris et Windows."

Perl2Exe (<http://www.indigostar.com/perl2exe.htm>) est un programme en ligne de commande qui convertit les scripts perl en exécutables. Il existe aussi bien pour Windows que pour les plateformes unix.

### 20.1.22 Comment compiler Perl pour en faire du Java ?

Vous pouvez intégrer Java et Perl avec le Perl Resource Kit d'O'Reilly Media. Voir <http://www.oreilly.com/catalog/prkunix/>.

Perl 5.6 vient avec Java Perl Lingo (ou JPL). JPL, qui est encore en développement, permet d'appeler du code Perl depuis Java. Lire le fichier jpl/README dans l'arborescence des sources Perl.

### 20.1.23 Comment faire fonctionner #!perl sur [MS-DOS,NT,...] ?

Pour OS/2 utilisez juste

```
extproc perl -S -your_switches
```

comme première ligne dans le fichier \*.cmd (-S est dû à un bug dans la gestion de "extproc" par cmd.exe). Pour DOS, il faudrait d'abord que quelqu'un crée un fichier batch similaire puis le codifie dans ALTERNATIVE\_SHEBANG (voir le fichier *INSTALL* dans la distribution pour plus d'informations).

L'installation sur Win95/98/NT, avec le portage Perl de ActiveState, va modifier la table de registre pour associer l'extension .pl avec l'interpréteur perl. Si vous utilisez un autre portage, peut-être même en compilant votre propre Perl Win95/98/NT à l'aide d'une version Windows de gcc (e.g. avec cygwin ou mingw32), alors vous devrez modifier la base de registres vous-même. En plus d'associer .pl avec l'interpréteur, les gens sous NT peuvent utiliser SET PATHEXT=%PATHEXT%; .PL pour qu'ils puissent exécuter le programme *install-linux.pl* en tapant simplement *install-linux*.

Sous "Classic" ou MacOS, les programmes Perl auront les attributs Créateur et Type appropriés, un double-clic dessus appellera donc l'application Perl. Sous Mac OS X, les scripts utilisant une ligne #! . . . peuvent être transformés en vraies applications via l'utilitaire DropScript de Wil Sanchez (<http://www.wsanchez.net/software/>).

**IMPORTANT!** : Quoi que vous fassiez, SURTOUT ne vous contentez pas seulement d'installer votre interpréteur dans votre répertoire cgi-bin, pour faire fonctionner votre serveur web avec vos programmes. C'est un ÉNORME risque de sécurité. Prenez le temps de bien vérifier que tout fonctionne correctement.

### 20.1.24 Puis-je écrire des programmes Perl pratiques sur la ligne de commandes ?

Oui. Lisez *perlrun* pour plus d'informations. Voici quelques exemples. (on considère ici un shell Unix avec les règles standard d'apostrophes.)

```
Additionner le premier et lz dernier champs
perl -lane 'print $F[0] + $F[-1]' *
```

```
Identifier des fichiers-textes
perl -le 'for(@ARGV) {print if -f && -T _}' *

enlever la plupart des commentaires d'un programme C
perl -0777 -pe 's/{/*.*?*/}gs' foo.c

Rajeunir un fichier d'un mois
perl -e '$X=24*60*60; utime(time(),time() + 30 * $X,@ARGV)' *

Trouver le premier uid non utilisé
perl -le '$i++ while getpwuid($i); print $i'

Afficher des chemins raisonnables vers des répertoires man
echo $PATH | perl -nl -072 -e '
 s![^/+]*$!man!&&-d&&!$s{$_}++&&push@m,$_;END{print"@m"}'
```

OK, le dernier n'est pas très simple. :-)

### 20.1.25 Pourquoi les commandes Perl à une ligne ne fonctionnent-elles pas sur mon DOS/Mac/VMS ?

Le problème est généralement que les interpréteurs de commandes sur ces systèmes ont des points de vue différents sur les apostrophes, guillemets, etc, par rapport aux shell Unix sous lesquels a été créée cette possibilité de commande à une ligne. Sur certains systèmes, vous devrez changer les apostrophes en guillemets, ce que vous ne devez *PAS* faire sur Unix ou sur des systèmes Plan9. Vous devrez aussi probablement changer un simple % en %%.

Par exemple :

```
Unix
perl -e 'print "Hello world\n"'

DOS, etc.
perl -e "print \"Hello world\n\""

Mac
print "Hello world\n"
(then Run "Myscript" or Shift-Command-R)

MPW
perl -e 'print "Hello world\n"'

VMS
perl -e "print ""Hello world\n"""
```

Le problème est que rien de tout cela n'est garanti : cela dépend de l'interpréteur de commande. Sous Unix, les deux premiers exemples marchent presque toujours. Sous DOS il est bien possible qu'aucun d'entre eux ne fonctionne. Si 4DOS est l'interpréteur de commandes, vous aurez probablement plus de chances avec ceci :

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>"""
```

Sous Mac, cela dépend de l'environnement que vous utilisez. Le shell MacPerl ou MPW, ressemble plutôt aux shells Unix car il accepte pas mal de variantes dans les apostrophes, guillemets, etc, excepté qu'il utilise librement les caractères de contrôle Mac non ASCII comme des caractères normaux.

L'usage de qq(), q() et qx(), à la place de "guillemets", d'apostrophes' et d'«accents graves» peut rendre les programmes sur une ligne plus faciles à écrire.

Il n'y a pas de solution globale à tout cela. Il y a un manque.

[Kenneth Albanowski a contribué à certaines de ces réponses.]

### 20.1.26 Où puis-je en apprendre plus sur la programmation CGI et Web en Perl ?

Pour les modules, prenez les modules CGI ou LWP au CPAN. Pour les bouquins, voyez les deux tout spécialement dédiés au développement pour le web, dans la question sur les livres. Pour des problèmes ou questions du style "Pourquoi ai-je une erreur 500" ou "Pourquoi cela ne fonctionne-t-il pas bien par le navigateur alors que tout marche depuis la ligne de commande shell", lisez <perlfaq9> ou la MetaFAQ CGI :

`http://www.perl.org/CGI_MetaFAQ.html`

### 20.1.27 Où puis-je apprendre la programmation orientée objet en Perl ?

Un bon point de départ est *perltoot* puis ensuite vous pouvez utiliser *perlobj*, *perlboot*, *perltooc* et *perlbot* comme références.

Deux bons livres sur la programmation orientée objet en Perl sont "Object-Oriented Perl" de Damian Conway chez Manning et "Learning Perl! References, Objects, & Modules" par Randal Schwartz et Tom Phoenix chez O'Reilly Media.

### 20.1.28 Où puis-je en apprendre plus sur l'utilisation liée de Perl et de C ?

Si vous voulez appeler du C à partir du Perl, commencez avec *perlxtst*, puis *perlx*, *xsubpp*, et *perlguts*. Si vous voulez appeler du Perl à partir du C, alors lisez *perlembd*, *perlcall*, et *perlguts*. N'oubliez pas que vous pouvez apprendre beaucoup en regardant comment des auteurs de modules d'extension ont écrit leur code et résolu leurs problèmes.

Vous n'avez peut-être pas besoin de toute la puissance de XS. Le module `Inline::C` vous permet de placer du code C directement dans votre script Perl. Il gère lui-même toute la magie nécessaire au bon fonctionnement. Vous devez encore connaître un minimum de choses sur l'API Perl mais vous n'avez plus à gérer toute la complexité des fichiers XS.

### 20.1.29 J'ai lu *perlembd*, *perlguts*, etc., mais je ne peux inclure du perl dans mon programme C, qu'est ce qui ne va pas ?

Téléchargez le kit `ExtUtils::Embed` depuis CPAN et exécutez `'make test'`. Si le test est bon, lisez les pods encore et encore et encore. Si le test échoue, voyez *perlbug* et envoyez un rapport de bug avec les sorties écran de `make test TEST_VERBOSE=1` ainsi que de `perl -V`.

### 20.1.30 Quand j'ai tenté d'exécuter mes scripts, j'ai eu ce message. Qu'est ce que cela signifie ?

Une liste complète des messages d'erreur et des avertissements de Perl accompagnés d'un texte explicatif se trouve dans *perldiag*. Vous pouvez aussi utiliser le programme `splain` (distribué avec perl) pour expliquer les messages d'erreur :

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

ou modifiez votre programme pour qu'il vous explique les messages :

```
use diagnostics;
```

ou

```
use diagnostics -verbose;
```

### 20.1.31 Qu'est-ce que `MakeMaker` ?

Ce module (qui fait partie de la distribution standard de Perl) est fait pour écrire un Makefile pour un module d'extension à partir d'un `Makefile.PL`. Pour plus d'informations, voyez *ExtUtils::MakeMaker*.

## 20.2 AUTEUR ET COPYRIGHT

Copyright (c) 1997-1999 Tom Christiansen et Nathan Torkington. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendante de cette distribution, tous les codes d'exemple ici, sont du domaine public. Vous êtes autorisé et encouragé à les utiliser tels quels ou de façon dérivée dans vos propres programmes, pour le fun ou pour le profit, comme vous le voulez. Un simple commentaire signalant les auteurs serait bien courtois, mais n'est pas obligatoire.

## 20.3 TRADUCTION

### 20.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 20.3.2 Traducteur

Sébastien Joncheray <info@raynette.com>. Mise à jour : Paul Gaborit <paul.gaborit at enstimac.fr>.

### 20.3.3 Relecture

Pascal Ethvignot <pascal@encelade.frmug.org>, Roland Trique <roland.trique@uhb.fr>, Gérard Delafond.

# Chapitre 21

## perlfaq4

Manipulation de données

### 21.1 DESCRIPTION

Cette section de la FAQ répond aux questions liées à la manipulation des nombres, des dates, des chaînes de caractères, des tableaux, des tables de hachage, ainsi qu'à divers problèmes relatifs aux données.

### 21.2 Données : nombres

#### 21.2.1 Pourquoi est-ce que j'obtiens des longs nombres décimaux (ex. 19.9499999999999) à la place du nombre que j'attends (ex. 19.95) ?

En interne, un ordinateur représente les flottants (les nombres à virgule) sous forme binaire. Les ordinateurs ne peuvent pas stocker tous les nombres de manière exacte. Certains nombres réels perdent un peu de précision lors de leur traitement. C'est un problème dû à la manière dont les ordinateurs stockent les nombres et concerne donc tous les langages, pas uniquement Perl.

Voir *perlnumber* pour de plus amples informations sur le stockage des nombres et leurs conversions.

Pour limiter le nombre de décimales dans un nombre, vous pouvez utiliser les fonctions `printf` ou `sprintf`. Voir `perlop` (§??) pour plus de détails.

```
printf "%.2f", 10/3;

my $number = sprintf "%.2f", 10/3;
```

#### 21.2.2 Pourquoi `int()` ne fonctionne pas bien ?

Votre fonction `int()` fonctionne certainement très bien. Ce sont les nombres qui ne sont pas ce que vous croyez.

Tout d'abord, voyez la question ci-dessus `perlfaq4` (§21.2.1).

Par exemple, ceci :

```
print int(0.6/0.2-2), "\n";
```

affichera 0 et non 1 sur la plupart des ordinateurs, puisque même des nombres aussi simples que 0.6 et 0.2 ne peuvent pas être représentés exactement par des flottants. Ce que vous espériez être 'trois' ci-dessus et en fait quelque chose comme 2.9999999999999995559.

### 21.2.3 Pourquoi mon nombre octal n'est-il pas interprété correctement ?

Perl ne comprend les nombres octaux et hexadécimaux en tant que tels que lorsqu'ils sont utilisés comme des valeurs littérales dans le programme. En Perl, les nombres octaux littéraux doivent commencer par "0" et les hexadécimaux par "0x". S'ils sont lus de quelque part et affectés à une variable, aucune conversion automatique n'a lieu. Pour convertir les valeurs, il faut utiliser explicitement oct() ou hex(). oct() interprète à la fois les nombres hexadécimaux ("0x350"), les nombres octaux ("0350" ou même sans le "0" de tête, comme dans "377") et les nombres binaires ("0b1010"), tandis que hex() ne convertit que les hexadécimaux, avec ou sans l'en-tête "0x", comme pour "0x255", "3A", "ff", ou "baffe". La transformation inverse (depuis décimal vers octal) peut être obtenue via les formats "%o" et "%O" de sprintf().

Ce problème apparaît fréquemment lorsque l'on essaye d'utiliser les fonctions chmod(), mkdir(), umask(), ou sysopen() qui demandent toutes traditionnellement des permissions en octal.

```
chmod(644, $file); # FAUX
chmod(0644, $file); # correct
```

Notez l'erreur de la première ligne qui spécifie le nombre décimal 644 au lieu du nombre octal voulu 0644. Le problème apparaît mieux avec :

```
printf("%#o", 644); # affiche 01204
```

Vous ne vouliez pas dire chmod(01204, \$file)... Si vous voulez utiliser des valeurs numériques octales comme arguments de chmod() et autres, alors exprimez les en les préfixant d'un zéro et en n'utilisant ensuite que des chiffres de 0 à 7.

### 21.2.4 Perl a-t-il une fonction round() ? Et ceil() (majoration) et floor() (minoration) ? Et des fonctions trigonométriques ?

Il faut retenir que int() ne fait que tronquer vers 0. Pour arrondir à un certain nombre de chiffres, sprintf() ou printf() sont d'habitude la voie la plus simple.

```
printf("%.3f", 3.1415926535); # affiche 3.142
```

Le module POSIX (élément de la distribution standard de Perl) implémente ceil(), floor(), et d'autres fonctions mathématiques et trigonométriques.

```
use POSIX;
$ceil = ceil(3.5); # 4
$floor = floor(3.5); # 3
```

Dans les versions 5.000 à 5.003 de perl, la trigonométrie était faite par le module Math::Complex. À partir de 5.004, le module Math::Trig (élément de la distribution standard de Perl) implémente les fonctions trigonométriques. En interne, il utilise le module Math::Complex, et quelques fonctions peuvent s'échapper de l'axe des réels vers le plan des complexes, comme par exemple le sinus inverse de 2.

Les arrondis dans des applications financières peuvent avoir des conséquences majeures, et la méthode utilisée se doit d'être spécifiée précisément. Dans ces cas, il vaut mieux ne pas avoir confiance dans un quelconque système d'arrondis utilisé par Perl, mais implémenter sa propre fonction d'arrondis telle qu'elle est nécessaire.

Pour comprendre pourquoi, remarquez comment il vous reste un problème lors d'une progression par cinq centièmes :

```
for ($i = 0; $i < 1.01; $i += 0.05) { printf "%.1f ", $i}

0.0 0.1 0.1 0.2 0.2 0.2 0.3 0.3 0.4 0.4 0.5 0.5 0.6 0.7 0.7
0.8 0.8 0.9 0.9 1.0 1.0
```

Perl n'est pas en faute. C'est pareil qu'en C. L'IEEE dit que nous devons faire comme ça. Les nombres en Perl dont la valeur absolue est un entier inférieur à 2\*\*31 (sur les machines 32 bit) fonctionneront globalement comme des entiers mathématiques. Les autres nombres ne sont pas garantis.



### 21.2.5 Comment faire des conversions numériques entre différentes bases, entre différentes représentations ?

Comme d'habitude avec Perl, il y a plusieurs moyens de le faire. Ci-dessous, vous trouverez divers exemples d'approches permettant de faire les conversions courantes entre les représentations des nombres. Ce n'est pas une liste exhaustive.

Certains exemples ci-dessous utilisent le module `Bit::Vector` du CPAN. Les raisons du choix de `Bit::Vector` plutôt que des fonctions prédéfinies de perl sont qu'il sait traiter des nombres de n'importe quelle longueur, qu'il est optimisé pour certaines opérations et que sa notation semblera familière au moins à certains programmeurs.

#### Comment faire une conversion hexadécimal vers décimal

En utilisant la conversion interne de perl de la notation `0x` :

```
$dec = 0xDEADBEEF;
```

En utilisant la fonction `hex` :

```
$dec = hex("DEADBEEF");
```

En utilisant `pack` :

```
$dec = unpack("N", pack("H8", substr("0" x 8 . "DEADBEEF", -8)));
```

En utilisant le module `Bit::Vector` du CPAN :

```
use Bit::Vector;
$vec = Bit::Vector->new_Hex(32, "DEADBEEF");
$dec = $vec->to_Dec();
```

#### Comment faire une conversion décimal vers hexadécimal

En utilisant `sprintf` :

```
$hex = sprintf("%X", 3735928559); # majuscules A-F
$hex = sprintf("%x", 3735928559); # minuscules a-f
```

En utilisant `unpack` :

```
$hex = unpack("H*", pack("N", 3735928559));
```

En utilisant `Bit::Vector` :

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$hex = $vec->to_Hex();
```

En utilisant `Bit::Vector` qui accepte un nombre de bits impair :

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(33, 3735928559);
$vec->Resize(32); # suppression des 0 de remplissage
$hex = $vec->to_Hex();
```

#### Comment faire une conversion octal vers décimal

En utilisant la conversion interne de Perl des nombres préfixés par un zéro :

```
$dec = 033653337357; # notez le préfixe 0 !
```

En utilisant la fonction `oct` :

```
$dec = oct("33653337357");
```

En utilisant `Bit::Vector`:

```
use Bit::Vector;
$vec = Bit::Vector->new(32);
$vec->Chunk_List_Store(3, split(//, reverse "33653337357"));
$dec = $vec->to_Dec();
```

#### Comment faire une conversion décimal vers octal

En utilisant `sprintf` :

```
$oct = sprintf("%o", 3735928559);
```

En utilisant Bit::Vector:

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$oct = reverse join(' ', $vec->Chunk_List_Read(3));
```

### Comment faire une conversion binaire vers décimal

Depuis Perl 5.6, vous pouvez écrire les nombres binaires directement via la notation 0b :

```
$number = 0b10110110;
```

En utilisant oct :

```
my $input = "10110110";
$decimal = oct("0b$input");
```

En utilisant pack et ord :

```
$decimal = ord(pack('B8', '10110110'));
```

En utilisant pack et unpack pour les grands nombres :

```
$int = unpack("N", pack("B32",
 substr("0" x 32 . "1111010101101101111011101111", -32)));
$dec = sprintf("%d", $int);
```

```
substr() est utilisée pour compléter le nombre par des zéros
à gauche afin d'obtenir une chaîne de 32 caractères
```

En utilisant Bit::Vector :

```
$vec = Bit::Vector->new_Bin(32, "1101111010101101101111011101111");
$dec = $vec->to_Dec();
```

### Comment faire une conversion décimal vers binaire

En utilisant sprintf (perl 5.6 et plus) :

```
$bin = sprintf("%b", 3735928559);
```

En utilisant unpack :

```
$bin = unpack("B*", pack("N", 3735928559));
```

En utilisant Bit::Vector :

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$bin = $vec->to_Bin();
```

Les autres conversions (par exemple hex -> oct, bin -> hex, etc.) sont laissées en exercices au lecteur.

## 21.2.6 Pourquoi & ne fonctionne-t-il pas comme je le veux ?

Le comportement des opérateurs arithmétiques binaires varie selon qu'ils sont utilisés sur des nombres ou des chaînes. Ces opérateurs traitent une chaîne comme une série de bits et travaillent avec (la chaîne "3" est le motif de bits 00110011). Ces opérateurs travaillent avec la forme binaire d'un nombre (le nombre 3 est traité comme le motif de bits 00000011).

Le fait de dire 11 & 3 effectue donc l'opération "et" bit-à-bit sur des nombres (donnant ici 3). Le fait de dire "11" & "3" effectue un "et" bit-à-bit sur des chaînes (donnant "1").

La plupart des problèmes posés par & et | surviennent parce que le programmeur pense qu'il traite un nombre alors qu'en fait c'est une chaîne. Les autres problèmes se posent parce que le programmeur dit :

```
if ("\020\020" & "\101\101") {
 # ...
}
```

mais une chaîne constituée de deux octets nuls (le résultat de "\020\020" & "\101\101") n'est pas une valeur fausse en Perl. Vous avez besoin d'écrire :

```
if (("\020\020" & "\101\101") !~ /^[^000]/) {
 # ...
}
```

### 21.2.7 Comment multiplier des matrices ?

Utiliser les modules `Math::Matrix` ou `Math::MatrixReal` (disponibles sur le CPAN) ou l'extension PDL (également disponible sur le CPAN).

### 21.2.8 Comment effectuer une opération sur une série d'entiers ?

Pour appeler une fonction sur chaque élément d'un tableau, et récupérer le résultat, utiliser :

```
@results = map { my_func($_) } @array;
```

Par exemple :

```
@triple = map { 3 * $_ } @single;
```

Pour appeler une fonction sur chaque élément d'un tableau, mais sans tenir compte du résultat :

```
foreach $iterator (@array) {
 some_func($iterator);
}
```

Pour appeler une fonction sur chaque entier d'un (petit) intervalle, on **peut** utiliser :

```
@results = map { some_func($_) } (5 .. 25);
```

mais il faut être conscient que l'opérateur `..` crée un tableau de tous les entiers de l'intervalle utilisant beaucoup de mémoire pour de grands intervalles. Il vaut mieux utiliser :

```
@results = ();
for ($i=5; $i < 500_005; $i++) {
 push(@results, some_func($i));
}
```

Cette situation a été corrigée dans Perl5.005. L'usage de `..` dans une boucle `for` itérera sur l'intervalle, sans créer tout l'intervalle.

```
for my $i (5 .. 500_005) {
 push(@results, some_func($i));
}
```

ne créera pas une liste de 500 000 entiers.

### 21.2.9 Comment produire des chiffres romains ?

Récupérez et utilisez le module <http://www.cpan.org/modules/by-module/Roman>.

### 21.2.10 Pourquoi mes nombres aléatoires ne sont-ils pas aléatoires ?

Si vous utilisez une version de Perl antérieure à la 5.004, vous devez appeler `srand` une fois au début de votre programme pourensemencer le générateur de nombres aléatoires.

```
BEGIN { srand() if $] < 5.004 }
```

Les versions 5.004 et supérieures appellent automatiquement `srand` dès le début. N'appellez pas `srand` plus d'une fois – vous rendriez vos nombres moins aléatoires, plutôt que l'inverse.

Les ordinateurs sont doués pour être déterministes et mauvais lorsqu'il s'agit d'être aléatoires (malgré les apparences provoquées par les bogues dans vos programmes :-). Lisez l'article *random* par Tom Phoenix dans la collection "Far More Than You Ever Wanted To Know" sur <http://www.cpan.org/misc/olddoc/FMTEYEWTk.tgz> pour en savoir plus à ce sujet. John von Neumann disait : « Quiconque tente de produire des nombres aléatoires par des moyens déterministes vit dans le péché. »

Si vous désirez des nombres qui soient plus aléatoires que ce que fournit `rand` avec `srand`, jetez aussi un oeil au module `Math::TrulyRandom` sur le CPAN. Il utilise des imperfections de l'horloge du système pour générer des nombres aléatoires, mais ceci prend un certain temps. Si vous voulez un meilleur générateur pseudo-aléatoire que celui qui vient avec le système d'exploitation, lisez les « Numerical Recipes in C » sur <http://www.nr.com/>.

### 21.2.11 Comment obtenir un nombre aléatoire entre X et Y ?

`rand($x)` retourne un nombre tel que  $0 \leq \text{rand}(\$x) < \$x$ . Donc perl peut générer un nombre aléatoire entre 0 et la valeur qui sépare X et Y.

Et donc, pour obtenir un entier entre 10 et 15 inclus, vous pouvez générer un nombre entre 0 et 5 auquel vous ajouterez 10.

```
my $nombre = 10 + int rand(15-10+1);
```

Ensuite, on peut transformer ce calcul pour en faire une fonction générique. Elle choisit un nombre aléatoire entre deux entiers donnés :

```
sub random_int_in ($$) {
 my($min, $max) = @_;
 # On suppose que les deux arguments sont eux-mêmes des entiers !
 return $min if $min == $max;
 ($min, $max) = ($max, $min) if $min > $max;
 return $min + int rand(1 + $max - $min);
}
```

## 21.3 Données : dates

### 21.3.1 Comment trouver le jour ou la semaine de l'année ?

La fonction `localtime()` retourne le jour de l'année. Appeler sans argument, `localtime` se base sur l'heure (et la date) courante.

```
$jour_de_l_annee = (localtime)[7];
```

Le module POSIX propose le jour ou la semaine de l'année dans les formats disponible :

```
use POSIX qw/strftime/;
my $jour_de_l_annee = strftime "%j", localtime;
my $semaine_de_l_annee = strftime "%W", localtime;
```

Pour obtenir le jour de l'année d'une date quelconque, utilisez le module `Time::Local` pour obtenir un temps en secondes depuis l'origine des temps que vous fournirez comme argument à `localtime`.

```
use POSIX qw/strftime/;
use Time::Local;
my $semaine_de_l_annee = strftime "%W",
 localtime(timelocal(0, 0, 0, 18, 11, 1987));
```

Le module `Date::Calc` propose deux fonctions pour calculer cela :

```
use Date::Calc;
my $jour_de_l_annee = Day_of_Year(1987, 12, 18);
my $semaine_de_l_annee = Week_of_Year(1987, 12, 18);
```

### 21.3.2 Comment trouver le siècle ou le millénaire actuel ?

Utilisez les fonctions simples suivantes :

```
sub trouve_siecle {
 return int((((localtime(shift || time))[5] + 1999))/100);
}

sub trouve_millenaire {
 return 1+int((((localtime(shift || time))[5] + 1899))/1000);
}
```

Sur certains systèmes, la fonction `strftime()` du module POSIX a été étendue d'une façon non standard pour utiliser le format `%C`, et ils prétendent que cela représente le "siècle". Ce n'est pas le cas, puisque sur la plupart de ces systèmes, cela ne représente que les deux premiers chiffres de l'année à quatre chiffres, et ne peut ainsi pas être utilisé pour déterminer de façon fiable le siècle ou le millénaire courant.

### 21.3.3 Comment comparer deux dates ou en calculer la différence ?

(contribution de brian d foy)

Vous pouvez tout simplement stocker vos dates sous formes de nombres et les soustraire. Mais les choses ne sont pas toujours aussi simples. Si vous souhaitez utiliser des dates mises en forme, les modules `Date::Manip`, `Date::Calc` ou `DateTime` devraient vous aider.

### 21.3.4 Comment convertir une chaîne de caractères en secondes depuis l'origine des temps ?

Si cette chaîne est suffisamment régulière pour avoir toujours le même format, on peut la découper et en passer les morceaux à la fonction `timelocal` du module standard `Time::Local`. Autrement, regarder les modules `Date::Calc` et `Date::Manip` disponibles sur le CPAN.

### 21.3.5 Comment trouver le jour du calendrier Julien ?

(contribution de brian d foy et de Dave Cross)

Vous pouvez utiliser le module `Time::JulianDay` disponible sur le CPAN. Assurez-vous tout de même que c'est bien le jour *Julien* que vous voulez. Chacun ayant son idée sur les jours Julien, regardez [http://www.hermetic.ch/cal\\_stud/jdn.htm](http://www.hermetic.ch/cal_stud/jdn.htm) pour en savoir plus.

Vous pouvez aussi essayer le module `DateTime` qui sait convertir une date ou une heure en un jour Julien.

```
$ perl -MDateTime -le'print DateTime->today->jd'
2453401.5
```

Ou le jour Julien modifié

```
$ perl -MDateTime -le'print DateTime->today->mjd'
53401
```

Ou même le jour de l'année (que certains croient être le jour Julien)

```
$ perl -MDateTime -le'print DateTime->today->doy'
31
```

### 21.3.6 Comment trouver la date d'hier ?

(contribution de brian d foy)

Utilisez l'un des modules `Date`. Le module `DateTime` fait cela simplement et vous donne le même moment (même heure) mais hier.

```
use DateTime;

my $hier = DateTime->now->subtract(days => 1);

print "Hier était $hier\n";
```

Vous pouvez aussi choisir le module `Date::Calc` en utilisant sa fonction `Today_and_Now`.

```
use Date::Calc qw(Today_and_Now Add_Delta_DHMS);

my @hier = Add_Delta_DHMS(Today_and_Now(), -1, 0, 0, 0);

print "@hier\n";
```

De nombreuses personnes essaient d'utiliser le temps plutôt que le calendrier pour gérer les dates en supposant que tous les jours ont une durée de vingt quatre heures. Or pour de nombreuses personnes, ils existent deux jours par an où ce n'est pas vrai : les jours de changement d'heure (l'heure d'été et l'heure d'hiver). Laissez donc les modules faire ce boulot.

### 21.3.7 Perl a-t-il un problème avec l'an 2000 ? Perl est-il compatible an 2000 ?

Réponse courte : Non, Perl n'a pas de problème avec l'an 2000. Oui, Perl est compatible an 2000 (si ceci veut dire quelque chose). Toutefois, les programmeurs que vous avez embauchés pour l'utiliser ne le sont probablement pas.

Réponse longue : la question demande une vraie compréhension du problème. Perl est juste tout aussi compatible an 2000 que votre crayon – ni plus ni moins. Pouvez-vous utiliser votre crayon pour rédiger un mémo non compatible an 2000 ? Bien sûr que oui. Est-ce la faute du crayon ? Bien sûr que non.

Les fonctions de date et d'heure fournies avec Perl (`gmtime` et `localtime`) donnent des informations adéquates pour déterminer l'année bien au-delà de l'an 2000 (2038 marque le début des problèmes pour les machines 32-bits). L'année renvoyée par ces fonctions lorsqu'elles sont utilisées dans un contexte de liste est l'année, moins 1900. Pour les années entre 1910 et 1999 ceci est un nombre à deux chiffres *par hasard*. Pour éviter le problème de l'an 2000, ne traitez tout simplement pas l'année comme un nombre à deux chiffres. Elle ne l'est pas.

Quand `gmtime()` et `localtime()` sont utilisées dans un contexte scalaire, elles renvoient une chaîne qui contient l'année écrite en entier. Par exemple, `$timestamp = gmtime(1005613200)` fixe `$timestamp` à la valeur "Tue Nov 13 01:00:00 2001". Ici encore, il n'y a pas de problème de l'an 2000.

Ceci ne veut pas dire que Perl ne peut pas être utilisé pour créer des programmes qui ne respecteront pas l'an 2000. Il peut l'être. Mais un crayon le peut aussi. La faute en revient à l'utilisateur, pas au langage. Au risque d'indisposer la NRA (National Rifle Association) : « Perl ne viole pas l'an 2000, les gens le font ». Pour un exposé plus complet, voir <http://www.perl.org/about/y2k.html>.

## 21.4 Données : chaînes de caractères

### 21.4.1 Comment m'assurer de la validité d'une entrée ?

(contribution de brian d foy)

Il existe plusieurs moyens de vérifier que les valeurs sont bien celles que vous attendez ou que vous acceptez. En plus des exemples spécifiques que vous trouverez dans la FAQ perl, vous pouvez aussi jeter un oeil aux modules dont le nom contient "Assert" ou "Validate" ainsi que d'autres modules comme `Regexp::Common`.

Quelques modules savent valider des types particuliers d'entrées. Par exemple `Business::ISBN`, `Business::CreditCard`, `Email::Valid` et `Data::Validate::IP`.

### 21.4.2 Comment supprimer les caractères d'échappement d'une chaîne de caractères ?

Cela dépend de que vous entendez par "caractère d'échappement". Les caractères d'échappement d'URL sont traités dans `<perlfaq9>`. Les caractères de l'interpréteur de commandes avec des barres obliques inversées (`\`) sont supprimés par :

```
s/\\(.)/$1/g;
```

Ceci ne convertira pas les `"\n"` ou `"\t"` ni aucun autre caractère spécial.

### 21.4.3 Comment enlever des paires de caractères successifs ?

(contribution de brian d foy)

Vous pouvez utiliser l'opérateur de substitution pour trouver des couples de caractères identiques et les remplacer pour un seul caractère. Dans cette substitution, nous cherchons un caractère (`.`). Les parenthèses de mémorisation stockent le caractère reconnu dans la référence `\1` que nous utilisons immédiatement pour reconnaître le même caractère une seconde fois. Nous remplaçons ensuite ce couple de caractères par le caractère `$1` (qui est l'équivalent de `\1` mais dans la chaîne de remplacement).

```
s/(.)\1/$1/g;
```

On peut aussi utiliser l'opérateur de translittération, `tr///`. Dans cet exemple, la liste de recherche est vide mais nous ajoutons le modificateur `c` pour utiliser en fait le complémentaire de cette liste, c'est à dire tout. La liste de remplacement, elle aussi, ne contient rien et donc la translittération est une opération vide puisque nous n'effectuons aucun remplacement (ou plus exactement, nous remplaçons chaque caractère par lui-même). Par contre, comme nous avons ajouté le modificateur `s`, les caractères consécutifs identiques sont remplacés par une seule occurrence.

```
my $str = 'Haarlem'; # aux Pays-bas
$str =~ tr//cs; # Harlem, comme à New-York
```

### 21.4.4 Comment effectuer des appels de fonction dans une chaîne ?

(contribution de brian d foy)

Ceci est documenté dans *perlref* et, bien que ce ne soit pas la chose la plus facile à lire, ça marche. Dans chacun des exemples suivants, nous appelons la fonction à l'intérieur des accolades en utilisant le déréférencement d'une référence. Si nous avons plus d'un seul résultat, nous pouvons construire et déréférencer un tableau anonyme. Dans ce cas, nous appelons la fonction dans un contexte de liste :

```
print "The time values are @{ [localtime] }.\n";
```

Si nous voulons appeler la fonction dans un contexte scalaire, il nous faut faire un petit effort. Il est possible de placer n'importe quel code entre les accolades donc il suffit de placer un code qui retourne une référence à un scalaire :

```
print "La date est ${\ (scalar localtime) }.\n"
print "La date est ${ my $x = localtime; \ $x }.\n";
```

Si votre fonction retourne déjà une référence, vous n'avez pas besoin de la construire vous-même :

```
sub timestamp { my $t = localtime; \ $t }
print "La date est ${ timestamp() }.\n";
```

Le module *Interpolation* peut aussi faire un peu de magie pour vous. Vous pouvez spécifier un nom, dans notre cas *E*, liée à une table de hachage qui fera l'interpolation pour vous. Il y a encore plusieurs autres moyens de faire cela.

```
use Interpolation E => 'eval';
print "The time values are $E{localtime() }.\n";
```

Dans la plupart des cas, il est probablement plus simple d'utiliser la concaténation de chaîne qui impose un contexte scalaire :

```
print "La date est " . localtime . ".\n";
```

### 21.4.5 Comment repérer des éléments appariés ou imbriqués ?

Ceci ne peut être réalisé en une seule expression régulière, même très compliquée. Pour trouver quelque chose se situant entre deux caractères simples, un motif comme `/x([\^x]*)x/` mettra les morceaux de l'intervalle dans `$1`. Lorsque le séparateur est de plusieurs caractères, il faudrait en utiliser un ressemblant plus à `/alpha(.*)omega/`. Mais aucun de ces deux ne gère les motifs imbriqués. Pour les expressions imbriquées utilisant `(, {, C{}` ou `<` comme délimiteurs, utilisez le module *CPAN Regexp::Common* ou voyez `"{??{ code }}"` in *perlre*. Pour les autres cas, il vous faudra écrire un analyseur.

Si vous songez sérieusement à écrire un analyseur syntaxique, de nombreux modules ou gadgets pourront vous rendre la vie plus facile. Il y a les modules du *CPAN Parse::RecDescent*, *Parse::Yapp* et *Text::Balanced* ainsi que le programme *byacc*. Depuis *perl 5.8*, *Text::Balanced* fait partie de la distribution standard.

Une approche simple, mais destructive, consiste à partir de l'intérieur pour aller vers l'extérieur en retirant les plus petites parties imbriquées les unes après les autres :

```
while (s/BEGIN((?:?!BEGIN)(?!END).)*)END//gs) {
 # faire quelque chose de $1
}
```

Une approche plus complexe et contorsionnée est de faire faire le travail par le moteur d'expressions rationnelles de Perl. Ce qui suit est une courtoisie de Dean Inada, et a plutôt l'allure d'une entrée de l'*Obfuscated Perl Contest*, mais ce code fonctionne vraiment :

```
$_ contient la chaîne à analyser
BEGIN et END sont les marqueurs ouvrant et fermants pour le
texte inclus.

@C = ('(', '');
@) = (')', '');
($re=$_)=~s/((BEGIN)|(END)|.)/$)[!$3]\Q$1\E$([!$2]/gs;
@$ = (eval{/$re/}, $@!~/unmatched/i);
print join("\n", @$[0..$#$]) if($$[-1]);
```

### 21.4.6 Comment inverser une chaîne de caractères ?

Utiliser `reverse()` dans un contexte scalaire, tel qu'indiqué dans `reverse` in *perlfunc*.

```
$reversed = reverse $string;
```

### 21.4.7 Comment développer les tabulations dans une chaîne de caractères ?

On peut le faire soi-même :

```
1 while $string =~ s/\t+/' ' x (length($&) * 8 - length('$') % 8)/e;
```

Ou utiliser simplement le module `Text::Tabs` (qui fait partie de la distribution standard de Perl).

```
use Text::Tabs;
@expanded_lines = expand(@lines_with_tabs);
```

### 21.4.8 Comment remettre en forme un paragraphe ?

Utiliser `Text::Wrap` (qui fait partie de la distribution standard de Perl) :

```
use Text::Wrap;
print wrap("\t", ' ', @paragraphs);
```

Les paragraphes passés en argument à `Text::Wrap` ne doivent pas contenir de caractère de saut de ligne. `Text::Wrap` ne justifie pas les lignes (alignées à droite).

Ou utilisez le module CPAN `Text::Autoformat`. Les formatage de fichiers s'effectue aisément via un alias shell comme :

```
alias fmt="perl -i -MText::Autoformat -n0777 \
-e 'print autoformat $_, {all=>1}' $*"
```

Voir la documentation de `Text::Autoformat` pour connaître toutes ses possibilités.

### 21.4.9 Comment accéder à ou modifier N caractères d'une chaîne de caractères ?

Vous pouvez accéder aux premiers caractères d'une chaîne via `substr()`. Pour obtenir le premier caractère, par exemple, commencer à la position 0 et récupérer une sous-chaîne de longueur 1.

```
$string = "Just another Perl Hacker";
$first_char = substr($string, 0, 1); # 'J'
```

Pour modifier une partie d'une chaîne, vous pouvez utiliser le quatrième argument optionnel qui est la chaîne de remplacement.

```
substr($string, 13, 4, "Perl 5.8.0");
```

Vous pouvez aussi utiliser `substr()` en tant que lvalue :

```
substr($a, 0, 3) = "Tom";
```



### 21.4.10 Comment changer la nième occurrence de quelque chose ?

Il faut conserver soi-même l'évolution de `n`. Par exemple, si l'on souhaite changer la cinquième occurrence de "whoever" ou "whomever" en "whosoever" ou "whomsoever", sans tenir compte de la casse. Supposons dans la suite que `$_` contienne la chaîne à modifier.

```
$count = 0;
s{((whom?)ever)}{
 ++$count == 5 # Est-ce la cinquième ?
 ? "${2}soever" # oui: faire l'échange
 : $1 # non: le laisser tel quel
}ige;
```

Dans des cas plus généraux, on peut utiliser le modificateur `/g` dans une boucle `while`, en comptant le nombre de correspondances.

```
$WANT = 3;
$count = 0;
$_ = "One fish two fish red fish blue fish";
while (/(\w+)\s+fish\b/gi) {
 if (++$count == $WANT) {
 print "The third fish is a $1 one.\n";
 }
}
```

Ceci sort: "The third fish is a red one." On peut aussi utiliser un compteur de répétition, et un motif répété comme ceci :

```
/(?:\w+\s+fish\s+){2}(\w+)\s+fish/i;
```

### 21.4.11 Comment compter le nombre d'occurrences d'une sous-chaîne dans une chaîne de caractères ?

Plusieurs moyens sont possibles, avec une efficacité variable. Pour trouver le nombre d'un caractère particulier (X) dans une chaîne, on peut utiliser la fonction `tr///` comme ceci :

```
$string = "ThisXlineXhasXsomeXx'sXinXit";
$count = ($string =~ tr/X//);
print "There are $count X characters in the string";
```

Ceci marche bien lorsque l'on cherche un seul caractère. Cependant si l'on essaye de compter des sous-chaînes de plusieurs caractères à l'intérieur d'une chaîne plus grande, `tr///` ne marchera pas. On peut alors envelopper d'une boucle `while()` une recherche de motif globale. Par exemple, comptons les entiers négatifs :

```
$string = "-9 55 48 -2 23 -76 4 14 -44";
while ($string =~ /-\d+/g) { $count++ }
print "There are $count negative numbers in the string";
```

Une autre manière de faire utilise la reconnaissance globale dans un contexte de liste puis assigne le résultat à un scalaire, produisant ainsi le nombre de reconnaissances.

```
$count = () = $string =~ /-\d+/g;
```

### 21.4.12 Comment mettre en majuscule toutes les premières lettre des mots d'une ligne ?

Pour mettre en lettres majuscules toutes les premières lettres des mots :

```
$line =~ s/\b(\w)/\U$1/g;
```

Ceci a pour effet de bord de transformer "aujourd'hui" en "Aujourd'Hui". C'est parfois ce qu'on veut. Sinon, on peut préférer cette solution (suggérée par brian d foy) :

```
$string =~ s/ (
 (^\w) #au début d'une ligne
 | # ou
 (\s\w) #précédé d'un espace
)
 /\U$1/xg;
$string =~ /([\w']+)/\u\L$1/g;
```

Pour transformer toute la ligne en lettres majuscules :

```
$line = uc($line);
```

À l'inverse, pour avoir chaque mot en lettres minuscules, avec la première lettre majuscule :

```
$line =~ s/(\w+)/\u\L$1/g;
```

On peut (et l'on devrait toujours) rendre ces caractères sensibles aux locales en plaçant une directive `use locale` dans le programme. Voir *perllocale* pour d'interminables détails sur les locales.

On s'y réfère parfois comme consistant à mettre quelque chose en "casse de titre", mais ce n'est pas tout à fait juste. Observez la capitalisation correcte du film *Dr. Strangelove or : How I Learned to Stop Worrying and Love the Bomb*, par exemple. (NdT : d'autant que l'usage en français est de ne mettre de majuscule qu'en début de phrase et aux noms propres.)

Le module `Text::Autoformat` de Domian Conway fournit quelques transformations sympathiques :

```
use Text::Autoformat;
my $x = "Dr. Strangelove or: How I Learned to Stop ".
 "Worrying and Love the Bomb";

print $x, "\n";
for my $style (qw(sentence title highlight))
{
 print autoformat($x, { case => $style }), "\n";
}
```

### 21.4.13 Comment découper une chaîne séparée par un [caractère] sauf à l'intérieur d'un [caractère] ?

Plusieurs modules savent gérer ce genre d'analyses – `Text::Balanced`, `Text::CSV`, `Text::CSV_XS` et `Text::ParseWord` parmi d'autres.

Prenons l'exemple du cas où l'on souhaite découper une chaîne qui est subdivisée en différents champs par des virgules. On ne peut utiliser `split(/,/)` parce qu'il ne faudrait pas découper si la virgule est entre guillemets. Par exemple pour la ligne de donnée suivante :

```
SAR001,"","Cimetrix, Inc","Bob Smith","CAM",N,8,1,0,7,"Error, Core Dumped"
```

À cause de la restriction imposée par les guillemets, ceci devient un problème relativement complexe. Heureusement, nous avons Jeffrey Field, auteur du livre *Mastering Regular Expressions*, qui s'est penché sur le problème pour nous. Il suggère (en supposant que `$text` contient la chaîne) :

```

@new = ();
push(@new, $+) while $text =~ m{
 "([\^\\"\\]*(?:\\\[^\\"\\])*),"? # regroupe les éléments entre guillemets
 | ([^,]+),?
 | ,
}gx;
push(@new, undef) if substr($text,-1,1) eq ',';

```

Pour représenter un vrai guillemet à l'intérieur d'un champ délimité par des guillemets, il faut le protéger d'une barre oblique inverse comme caractère d'échappement (c.-à-d. "comme \"ceci\"").

Comme autre choix, le module `Text::ParseWords` (qui fait partie de la distribution standard de Perl) permet de faire :

```

use Text::ParseWords;
@new = quotewords(",", 0, $text);

```

Il existe aussi un module `Text::CSV` (Comma-Separated Values – Valeurs séparées par des virgules) sur le CPAN.

### 21.4.14 Comment supprimer des espaces blancs au début/à la fin d'une chaîne ?

(contribution de brian d foy)

Une substitution peut le faire pour vous. Pour une simple ligne, vous voulez remplacer tous les espaces en début et en fin de ligne par rien du tout. Cette double substitution le fait :

```

s/^\s+//;
s/\s+$//;

```

Vous pourriez aussi l'écrire en une seule substitution bien que ce soit plus lent. Mais ce n'est peut-être pas important pour vous.

```

s/^\s+|\s+$//g;

```

Dans cette expression rationnelle, l'alternative est reconnue soit en début soit en fin de chaîne puisque les ancres ont une précedence plus faible que l'alternative. Avec le modificateur `/g`, la substitution s'effectue partout où elle est reconnue, et donc au début et à la fin. Souvenez-vous que `\s+` reconnaît les caractères de fin de ligne et que `$` peut s'ancrer à la fin de la chaîne, donc le caractère de fin de ligne disparaîtra lui-aussi. Ajoutez juste le saut de ligne en sortie, ce qui a l'avantage de conserver les lignes entièrement blanches qui sont vidées par `^\s+`.

```

while(<>)
{
 s/^\s+|\s+$//g;
 print "$_\n";
}

```

Pour une chaîne multi-lignes, vous pouvez appliquer l'expression rationnelle à chaque ligne logique de la chaîne en ajoutant le modificateur `/m` (pour "multi-lignes"). Avec ce modificateur, `$` est reconnu *avant* un saut de ligne, et donc il n'est pas supprimé. Par contre, il supprimera encore le saut de ligne en fin de chaîne.

```

$string =~ s/^\s+|\s+$//gm;

```

Souvenez-vous que les lignes entièrement blanches vont disparaître puisque la première partie de l'alternative les remplacera par rien. Si vous souhaitez les lignes blanches, il faut faire un peu plus de travail. Au lieu de reconnaître n'importe quel espace (puisque cela inclut les sauts de ligne), ne reconnaissons que les autres espaces.

```

$string =~ s/^[\t\f]+|[\t\f]+$//mg;

```

### 21.4.15 Comment cadrer une chaîne avec des blancs ou un nombre avec des zéros ?

(Cette réponse est une contribution de Uri Guttman, avec un coup de main de Bart Lateur).

Dans les exemples suivants, `$pad_len` est la longueur dans laquelle vous voulez cadrer la chaîne, `$text` ou `$num` contient la chaîne à cadrer, et `$pad_char` contient le caractère de remplissage. Vous pouvez utiliser une constante contenant une chaîne d'un seul caractère à la place de la variable `$pad_char` si vous savez ce que c'est. Et de la même façon vous pouvez utiliser un entier à la place de `$pad_len` si vous connaissez à l'avance la longueur du cadrage.

La méthode la plus simple utilise la fonction `sprintf`. Elle peut cadrer à gauche et à droite avec des espaces et à gauche avec des zéros et elle ne tronquera pas le résultat. La fonction `pack` peut seulement cadrer des chaînes à droite avec des blancs et elle tronquera le résultat à la longueur maximale de `$pad_len`.

```
Cadrage à gauche d'une chaîne avec des blancs (pas de troncature)
$ padded = sprintf("%${pad_len}s", $text);
$ padded = sprintf("%*s", $pad_len, $text); # idem

Cadrage à droite d'une chaîne avec des blancs (pas de troncature)
$ padded = sprintf("%-${pad_len}s", $text);
$ padded = sprintf("%-*s", $pad_len, $text); # idem

Cadrage à gauche d'un nombre avec 0 (pas de troncature)
$ padded = sprintf("%0${pad_len}d", $num);
$ padded = sprintf("%0*d", $pad_len, $num); # idem

Cadrage à droite d'un nombre avec 0 (avec troncature)
$ padded = pack("A$pad_len", $text);
```

Si vous avez besoin de cadrer avec un caractère autre qu'un blanc ou un zéro, vous pouvez utiliser une des méthodes suivantes. Elles génèrent toutes une chaîne de cadrage avec l'opérateur `x` et la combinent avec `$text`. Ces méthodes ne tronquent pas `$text`.

Cadrage à gauche et à droite avec n'importe quel caractère, créant une nouvelle chaîne :

```
$ padded = $pad_char x ($pad_len - length($text)) . $text;
$ padded = $text . $pad_char x ($pad_len - length($text));
```

Cadrage à gauche et à droite avec n'importe quel caractère, modifiant directement `$text` :

```
substr($text, 0, 0) = $pad_char x ($pad_len - length($text));
$text .= $pad_char x ($pad_len - length($text));
```

### 21.4.16 Comment extraire une sélection de colonnes d'une chaîne de caractères ?

Utiliser les fonctions `substr()` ou `unpack()`, toutes deux documentées dans *perlfunc*. Ceux qui préfèrent penser en termes de colonnes plutôt qu'en longueurs de lignes peuvent utiliser ce genre de choses :

```
déterminer le format de unpack nécessaire pour découper la
sortie d'un ps de Linux
les arguments sont les colonnes sur lesquelles découper
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);
sub cut2fmt {
 my(@positions) = @_;
 my $template = '';
 my $lastpos = 1;
 for my $place (@positions) {
 $template .= "A" . ($place - $lastpos) . " ";
 $lastpos = $place;
 }
 $template .= "A*";
 return $template;
}
```

### 21.4.17 Comment calculer la valeur soundex d'une chaîne ?

(contribution de brian d foy)

Vous pouvez utiliser le module standard `Test::Soundex`. Si vous souhaitez faire de la reconnaissance floue et approchée, vous devriez aussi essayer les modules `String::Approx`, `Text::Metaphone` et `Text::DoubleMetaphone`.

### 21.4.18 Comment interpoler des variables dans des chaînes de texte ?

Supposons une chaîne comme celle-ci (\$bar et \$foo ne sont pas interpolées) :

```
$text = 'this has a $foo in it and a $bar';
```

Vous pouvez utiliser une substitution avec une double évaluation. Le premier `/e` remplace `$1` par `$foo` et le second `/e` remplace `$foo` par sa valeur. Vous pouvez habiller cela par un `eval` : si vous essayez d'obtenir la valeur d'une variable non déclaré et si `use strict` est actif, vous aurez une erreur fatale.

```
eval { $text =~ s/(\$w+)/$1/eeg };
die if $@;
```

Il serait probablement préférable dans le cas général de traiter ces variables comme des entrées dans une table de hachage à part. Par exemple :

```
%user_defs = (
 foo => 23,
 bar => 19,
);
$text =~ s/(\$w+)/$user_defs{$1}/g;
```

### 21.4.19 En quoi est-ce un problème de toujours placer "\$ vars" entre guillemets ?

Le problème est que ces guillemets forcent la conversion en chaîne, imposant aux nombres et aux références de devenir des chaînes, même lorsqu'on ne le souhaite pas. Pensez-y de cette façon : l'expansion des guillemets est utilisée pour produire de nouvelles chaînes. Si vous avez déjà une chaîne, pourquoi en vouloir plus ?

Si l'on prend l'habitude d'écrire des choses bizarres comme ça :

```
print "$var"; # MAL
$new = "$old"; # MAL
somefunc("$var"); # MAL
```

on se retrouve vite avec des problèmes. Les constructions suivantes devraient (dans 99.8% des cas) être plus simples et plus directes :

```
print $var;
$new = $old;
somefunc($var);
```

Autrement, en plus de ralentir, ceci risque de casser le code lorsque ce qui est dans la variable scalaire n'est effectivement ni une chaîne de caractères, ni un nombre mais une référence :

```
func(\@array);
sub func {
 my $aref = shift;
 my $oref = "$aref"; # FAUX
}
```

On peut aussi se retrouver face à des problèmes subtils pour les quelques opérations pour lesquels Perl fait effectivement la différence entre une chaîne de caractères et un nombre, comme pour l'opérateur magique d'autoincrément `++`, ou pour la fonction `syscall()`.

La mise en chaîne détruit aussi les tableaux :

```
@lines = 'command';
print "@lines"; # FAUX - ajoute des blancs
print @lines; # correct
```

### 21.4.20 Pourquoi est-ce que mes documents <<HERE ne marchent pas ?

Vérifier les trois points suivants :

**Il ne doit pas y avoir d'espace après le <<.**

**Il devrait (habituellement) y avoir un point-virgule à la fin.**

**On ne peut pas mettre (facilement) d'espace devant la marque.**

Si l'on souhaite indenter le texte du document inséré, on peut faire ceci :

```
tout à la fois
($VAR = <<HERE_TARGET) =~ s/^\s+//gm;
 your text
 goes here
HERE_TARGET
```

Mais la cible `HERE_TARGET` doit quand même être alignée sur la marge. Pour l'indenter également, il faut mettre l'indentation entre apostrophes.

```
($quote = <<' FINIS') =~ s/^\s+//gm;
 ...we will have peace, when you and all your works have
 perished--and the works of your dark master to whom you
 would deliver us. You are a liar, Saruman, and a corrupter
 of men's hearts. --Theoden in /usr/src/perl/taint.c
 FINIS
$quote =~ s/\s+---/\n--/;
```

Une jolie fonction d'usage général pour réarranger proprement des documents insérés indentés est donnée ci-dessous. Elle attend un document inséré en argument. Elle regarde si chaque ligne commence avec une sous-chaîne commune, et si tel est le cas, elle enlève cette sous-chaîne. Dans le cas contraire, elle prend le nombre d'espaces en tête de la première ligne et en enlève autant à chacune des lignes suivantes.

```
sub fix {
 local $_ = shift;
 my ($white, $leader); # espaces en commun et chaîne en commun
 if (/^\s*(?:([\w\s]+)\s*).*\n(?:\s*\1\2?.*\n)+$/) {
 ($white, $leader) = ($2, quotemeta($1));
 } else {
 ($white, $leader) = (/^\s+)/, ' ');
 }
 s/^\s*?$leader(?:$white)?//gm;
 return $_;
}
```

Ceci fonctionne avec des chaînes d'en-tête spéciales et déterminées dynamiquement :

```
$remember_the_main = fix<<' MAIN_INTERPRETER_LOOP';
 @@@ int
 @@@ runops() {
 @@@ SAVEI32(runlevel);
 @@@ runlevel++;
 @@@ while (op = (*op->op_ppaddr)());
 @@@ TAIN_T_NOT;
 @@@ return 0;
 @@@ }
MAIN_INTERPRETER_LOOP
```

Ou encore avec une quantité fixée d'en-tête d'espaces, tout en conservant correctement l'indentation :

```
$poem = fix<<EVER_ON_AND_ON;
 Now far ahead the Road has gone,
 And I must follow, if I can,
 Pursuing it with eager feet,
 Until it joins some larger way
 Where many paths and errands meet.
 And whither then? I cannot say.
 --Bilbo in /usr/src/perl/pp_ctl.c
EVER_ON_AND_ON
```

## 21.5 Données : tableaux

### 21.5.1 Quelle est la différence entre une liste et un tableau ?

Un tableau a une longueur variable. Une liste n'en a pas. Un tableau est quelque chose sur laquelle vous pouvez pousser ou retirer des données, alors qu'une liste est un ensemble de valeurs. Certaines personnes font la distinction qu'une liste est une valeur tandis qu'un tableau est une variable. Les sous-programmes se voient passer et renvoient des listes, vous mettez les choses dans un contexte de liste, vous initialisez des tableaux avec des listes, et vous balayez une liste avec `foreach()`. Les variables `@` sont des tableaux, les tableaux anonymes sont des tableaux, les tableaux dans un contexte scalaire se comportent comme le nombre de leurs éléments, les sous-programmes accèdent à leurs arguments via le tableau `@_` et les fonctions `push/pop/shift` ne fonctionnent que sur les tableaux.

Une note au passage : il n'existe pas de liste dans un contexte scalaire. Lorsque vous dites

```
$scalar = (2, 5, 7, 9);
```

vous utilisez l'opérateur virgule dans un contexte scalaire, c'est donc l'opérateur virgule scalaire qui est utilisé. Il n'y a jamais eu là de liste du tout ! C'est donc la dernière valeur qui est renvoyée : 9.

### 21.5.2 Quelle est la différence entre `$ array[1]` et `@array[1]` ?

La première est une valeur scalaire tandis que la seconde est une tranche de tableaux, ce qui en fait une liste avec une seule valeur (scalaire). On est censé utiliser `$` lorsque l'on désire une valeur scalaire (le plus souvent) et `@` lorsque l'on souhaite une liste avec une seule valeur scalaire à l'intérieur (très, très rarement, presque jamais en fait).

Il n'y a souvent pas de différence, mais il arrive que si. Par exemple, comparer :

```
$good[0] = 'une commande qui produit plusieurs lignes';
```

avec

```
@bad[0] = 'une commande qui produit plusieurs lignes';
```

La directive `use warnings` ou le drapeau `-w` préviennent lorsque de tels problèmes se présentent.

### 21.5.3 Comment supprimer les doublons d'une liste ou d'un tableau ?

(contribution de brian d foy)

Utilisez une table de hachage. Lorsque vous pensez à "unique" ou à "doublons", pensez aux "clés de hachage".

Si l'ordre des éléments ne compte pas, vous pouvez créer une table de hachage dont vous extrairez les clés. La manière de créer la table de hachage importe peu : il vous suffit d'utiliser `keys` pour récupérer les éléments uniques.

```
my %hachage = map { $_, 1 } @tableau;
ou via une tranche de hachage : @hachage{ @tableau } = ();
ou via un foreach : $hachage{$_} = 1 foreach (@tableau);

my @unique = keys %hachage;
```

Vous pouvez aussi parcourir tous les éléments et ne conserver que ceux que vous n'avez encore jamais vus. La première fois que la boucle traite un élément, cet élément n'a pas de clé correspondante dans %dejavu. La condition dans l'instruction commençant par `next` crée cette clé et récupère immédiatement sa valeur (qui est `undef`) avant de l'incrémenter. Comme cette valeur initiale est fautive, le `next` n'a pas lieu et la boucle continue en exécutant le `push`. Lorsque la boucle rencontre à nouveau le même élément, la clé correspondante existe dans la tableau de hachage *et* cette clé est vraie (puisque sa valeur incrémentée n'est plus `undef`), et donc le `next` interrompt cette itération et la boucle passe à l'élément suivant.

```
my @unique = ();
my %dejavu = ();

foreach my $elem (@tableau)
{
 next if $dejavu{ $elem }++;
 push @unique, $elem;
}
```

Vous pouvez écrire cela de manière plus concise en utilisant `grep`, qui fera la même chose.

```
my %dejavu = ();
my @unique = grep { ! $dejavu{ $_ }++ } @tableau;
```

#### 21.5.4 Comment savoir si une liste ou un tableau inclut un certain élément ?

(cette réponse est, en partie, une contribution de Anno Siegel)

D'entendre le mot "*inclut*" est une *indication* qu'on aurait dû utiliser un tableau de hachage, et pas une liste ou un tableau, pour enregistrer ses données. Les hachages sont conçus pour répondre rapidement et efficacement à cette question, alors que les tableaux ne le sont pas.

Cela dit il y a plusieurs moyens pour résoudre ce problème. Si vous faites cette requête plusieurs fois sur des valeurs de chaînes arbitraires, le plus rapide est probablement d'inverser le tableau original et de maintenir à jour une table de hachage dont les clefs sont les valeurs du tableau.

```
@blues = qw/azure cerulean teal turquoise lapis-lazuli/;
%is_blue = ();
for (@blues) { $is_blue{$_} = 1 }
```

Vous pouvez alors regarder si telle couleur (`$some_color`) est du bleu en testant `$is_blue{$some_color}`. Il aurait été une bonne idée de conserver les bleus dans un hachage dès le début.

Si les valeurs sont de petits entiers positifs, on peut simplement utiliser un tableau indexé. Ce genre de tableau prendra moins de place :

```
@primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
undef @is_tiny_prime;
for (@primes) { $is_tiny_prime[$_] = 1 }
ou simplement @istiny_prime[@primes] = (1) x @primes;
```

À partir de là pour vérifier si `$some_number` est un nombre premier il suffit de tester `$is_tiny_prime[$some_number]`.

Si les valeurs en question sont des entiers plutôt que des chaînes de caractères, on économise un certain espace en utilisant des chaînes de bits à la place :

```
@articles = (1..10, 150..2000, 2017);
undef $read;
for (@articles) { vec($read,$_,1) = 1 }
```

Et là vérifier si `vec($read,$n,1)` est vrai pour un `$n`.

Ces méthodes garantissent des test individuels rapides mais nécessitent une réorganisation de la liste ou du tableau original. Cela ne vaut le coup que si vous devez tester plusieurs valeurs du même tableau.

Si vous ne devez faire ce test qu'une seule fois, le module standard `List::Util` exporte la fonction `<first>`. Elle s'arrête dès qu'elle trouve l'élément recherché. Elle est écrite en C pour des raisons de performance et son équivalent en Perl ressemblerait au sous-programme suivant :



```

sub first (&@) {
 my $code = shift;
 foreach (@_) {
 return $_ if &{$code}();
 }
 undef;
}

```

Si la performance n'est pas un problème, une manière de faire consiste à appliquer à toutes la liste un `grep` dans un contexte scalaire (il retourne alors le nombre d'items ayant passé la condition). En revanche, vous y gagnez la possibilité de savoir combien de fois l'élément est reconnu.

```
my $nb_items = grep $_ eq $element, @tableau;
```

Si vous souhaitez récupérer les éléments reconnus, utilisez `grep` dans un contexte de liste.

```
my @reconnus = grep $_ eq $element, @tableau;
```

### 21.5.5 Comment calculer la différence entre deux tableaux ? Comment calculer l'intersection entre deux tableaux ?

Utiliser un hachage. Voici un code pour faire les deux et même plus. Il suppose chaque élément dans un tableau donné :

```

@union = @intersection = @difference = ();
%count = ();
foreach $element (@array1, @array2) { $count{$element}++ }
foreach $element (keys %count) {
 push @union, $element;
 push @{ $count{$element} > 1 ? \@intersection : \@difference }, $element;
}

```

Notez que ceci est la *différence symétrique*, c'est-à-dire tous les éléments qui sont soit dans A, soit dans B, mais pas dans les deux. Voyez cela comme une opération `xor` (un ou exclusif).

### 21.5.6 Comment tester si deux tableaux ou hachages sont égaux ?

Le code suivant fonctionne pour les tableaux à un seul niveau. Il utilise une comparaison de chaînes, et ne distingue pas les chaînes vides définies ou indéfinies. Modifiez-le si vous avez d'autres besoins.

```

$are_equal = compare_arrays(\@frogs, \@toads);

sub compare_arrays {
 my ($first, $second) = @_;
 no warnings; # silence spurious -w undef complaints
 return 0 unless @$first == @$second;
 for (my $i = 0; $i < @$first; $i++) {
 return 0 if $first->[$i] ne $second->[$i];
 }
 return 1;
}

```

Pour les structures à niveau multiples, vous pouvez désirer utiliser une approche ressemblant plus à celle-ci. Elle utilise le module CPAN `FreezeThaw` :

```

use FreezeThaw qw(cmpStr);
@a = @b = ("this", "that", ["more", "stuff"]);

```

```
printf "a and b contain %s arrays\n",
 cmpStr(\@a, \@b) == 0
 ? "the same"
 : "different";
```

Cette approche fonctionne aussi pour comparer des hachages. Ici, nous allons démontrer deux réponses différentes :

```
use FreezeThaw qw(cmpStr cmpStrHard);

%a = %b = ("this" => "that", "extra" => ["more", "stuff"]);
$a{EXTRA} = \%b;
$b{EXTRA} = \%a;

printf "a and b contain %s hashes\n",
 cmpStr(%a, %b) == 0 ? "the same" : "different";

printf "a and b contain %s hashes\n",
 cmpStrHard(%a, %b) == 0 ? "the same" : "different";
```

La première rapporte que les deux hachages contiennent les mêmes données, tandis que la seconde rapporte le contraire. Le fait de déterminer celle que vous préférez vous est laissée en exercice.

### 21.5.7 Comment trouver le premier élément d'un tableau vérifiant une condition donnée ?

Pour trouver le premier élément d'un tableau vérifiant une condition, vous pouvez utiliser la fonction `first` du module `List::Util` (en standard depuis Perl 5.8). L'exemple suivant trouve le premier élément contenant "Perl".

```
use List::Util qw(first);

my $element = first { /Perl/ } @tableau;
```

Si vous ne pouvez pas utiliser `List::Util`, vous pouvez écrire votre propre boucle pour faire la même chose. Dès que vous trouvez l'élément voulu, vous stoppez la boucle via `last()`.

```
my $found;
foreach (@tableau)
{
 if(/Perl/) { $found = $_; last }
}
```

Si vous souhaitez connaître l'indice de l'élément, vous pouvez faire une boucle sur tous les indices du tableau et tester l'élément lié à cet indice jusqu'à trouver l'élément satisfaisant la condition.

```
my($found, $index) = (undef, -1);
for($i = 0; $i < @tableau; $i++)
{
 if(tableau[$i] =~ /Perl/)
 {
 $found = tableau[$i];
 $index = $i;
 last;
 }
}
```

### 21.5.8 Comment gérer des listes chaînées ?

En général, avec Perl, on n'a pas de besoin de liste chaînée, puisqu'avec des tableaux usuels, on peut ajouter (push/unshift) et enlever (pop/shift) de part et d'autre, ou l'on peut utiliser splice pour ajouter et/ou enlever un nombre arbitraire d'éléments à un point arbitraire. Pop et shift sont toutes deux des opérations en  $O(1)$  sur les tableaux dynamiques de Perl. En absence de shifts et pops, push a en général besoin de faire des réallocations environ toutes les  $\log(N)$  fois, et unshift aura besoin de copier des pointeurs à chaque fois.

Si vous le voulez vraiment, vous pouvez utiliser les structures décrites dans *perldsc* ou *perltoot* et faire exactement comme le livre d'algorithmes dit de faire. Par exemple, imaginez un noeud de liste tel que celui-ci :

```
$node = {
 VALUE => 42,
 LINK => undef,
};
```

Vous pouvez balayer la liste de cette façon :

```
print "List: ";
for ($node = $head; $node; $node = $node->{LINK}) {
 print $node->{VALUE}, " ";
}
print "\n";
```

Vous pouvez allonger la liste ainsi :

```
my ($head, $tail);
$tail = append($head, 1); # ajoute un élément en tête
for $value (2 .. 10) {
 $tail = append($tail, $value);
}

sub append {
 my($list, $value) = @_;
 my $node = { VALUE => $value };
 if ($list) {
 $node->{LINK} = $list->{LINK};
 $list->{LINK} = $node;
 } else {
 $_[0] = $node; # remplace la valeur fournie à l'appel
 }
 return $node;
}
```

Mais encore une fois, les fonctions intégrées de Perl sont presque toujours suffisantes.

### 21.5.9 Comment gérer des listes circulaires ?

Des listes circulaires peuvent être gérées de façon traditionnelle par des listes chaînées, ou encore en utilisant simplement quelque chose comme ceci avec un tableau :

```
unshift(@array, pop(@array)); # les derniers seront les premiers
push(@array, shift(@array)); # et vice versa
```

### 21.5.10 Comment mélanger le contenu d'un tableau ?

Si vous disposez au moins de la version 5.8.0 de Perl ou si la version 1.03 ou plus de Scalar-List-Utils est installée, vous pouvez faire :

```
use List::Util 'shuffle';

@melange = shuffle(@tableau);
```

Sinon, utilisez le mélangeur de Fisher-Yates :

```
sub melange_de_fisher_yates {
 my $deck = shift; # $deck est une référence à un tableau
 my $i = @$deck;
 while (--$i) {
 my $j = int rand ($i+1);
 @$deck[$i,$j] = @$deck[$j,$i];
 }
}

mélange ma collection de mpeg
#
my @mpeg = <audio/*/*.mp3>;
melange_de_fisher_yates(\@mpeg); # mélange @mpeg sur place
print @mpeg;
```

Notez que le mélangeur ci-dessus mélange le tableau sur place alors que List::Util::shuffle() prend une liste et retourne une nouvelle liste mélangée.

Vous avez probablement vu des algorithmes de mélange qui fonctionnent en utilisant splice, choisissant au hasard un élément à mettre dans l'élément courant.

```
srand;
@new = ();
@old = 1 .. 10; # just a demo
while (@old) {
 push(@new, splice(@old, rand @old, 1));
}
```

Ceci est mauvais car splice est déjà en  $O(N)$ , et puisqu'on l'exécute  $N$  fois, on vient d'inventer un algorithme quadratique ; c'est-à-dire en  $O(N^2)$ . Ceci ne supporte pas la montée en charge, même si Perl est tellement efficace qu'on ne le remarquerait probablement pas avant d'atteindre des tableaux relativement grands.

### 21.5.11 Comment traiter/modifier chaque élément d'un tableau ?

Utiliser for/foreach :

```
for (@lines) {
 s/foo/bar/; # changer ce mot
 tr/XZ/ZX/; # échanger ces lettres
}
```

En voici un autre ; calculons des volumes sphériques :

```
for (@volumes = @radii) { # @volumes contient les parties modifiées
 $_ **= 3;
 $_ *= (4/3) * 3.14159; # ceci sera transformé en une constante
}
```

Ceci pourrait aussi être fait en utilisant map() qui permet de transformer une liste en une autre liste :

```
@volumes = map {$_ ** 3 * (4/3) * 3.14159} @radii;
```

Si vous souhaitez faire la même chose pour modifier les valeurs d'une table de hachage, vous pouvez utiliser la fonction `values`. Depuis perl 5.6, les valeurs ne sont pas copiées et donc, en modifiant `$orbit` (dans notre cas), vous modifiez réellement la valeur.

```
for $orbit (values %orbits) {
 ($orbit **= 3) *= (4/3) * 3.14159;
}
```

Dans les versions antérieures à perl 5.6, la fonction `values` retournaient des copies des valeurs et donc certains anciens codes peuvent encore contenir la construction `@orbits{keys %orbits}` à la place de `values %orbits`.

### 21.5.12 Comment sélectionner aléatoirement un élément d'un tableau ?

Utiliser la fonction `rand()` (voir `rand` in *perlfunc*) :

```
$index = rand @tableau;
$element = $tableau[$index];
```

Ou plus simplement :

```
my $element = $tableau[rand @tableau];
```

### 21.5.13 Comment générer toutes les permutations des N éléments d'une liste ?

Utilisez le module `List::Permutor` de CPAN. Si la liste est un véritable tableau, essayez le module `Algorithm::Permute` (disponible lui aussi dans CPAN). Il est écrit en code XS et est très efficace.

```
use Algorithm::Permute;
my @array = 'a'..'d';
my $p_iterator = Algorithm::Permute->new (\@array);
while (my @perm = $p_iterator->next) {
 print "next permutation: (@perm)\n";
}
```

Pour une exécution encore plus rapide, vous pouvez faire :

```
use Algorithm::Permute;
my @array = 'a'..'d';
Algorithm::Permute::permute {
 print "next permutation: (@array)\n";
} @array;
```

Voici un petit programme qui, à chaque ligne entrée, génère toutes les permutations de des mots de la ligne. L'algorithme utilisé dans la fonction `permute()` est présenté dans le quatrième volume (non encore publié) de *The Art of Computer Programming* par Knuth et fonctionnera pour n'importe quelle liste :

```
#!/usr/bin/perl -n
générateur de permutations ordonnées de Fischer-Kause

sub permute (&@) {
 my $code = shift;
 my @idx = 0..$#_;
 while ($code->(@_[@idx])) {
 my $p = $#idx;
 --$p while $idx[$p-1] > $idx[$p];
 my $q = $p or return;
 push @idx, reverse splice @idx, $p;
 ++$q while $idx[$p-1] > $idx[$q];
 @idx[$p-1,$q]=@idx[$q,$p-1];
 }
}

permute {print"@_\n"} split;
```

### 21.5.14 Comment trier un tableau par (n'importe quoi) ?

Fournir une fonction de comparaison à `sort()` (décrit dans `sort` in *perlfunc*) :

```
@list = sort { $a <=> $b } @list;
```

La fonction de tri par défaut est `cmp`, la comparaison des chaînes, laquelle trierait (1, 2, 10) en (1, 10, 2). `<=>`, utilisé ci-dessus, est l'opérateur de comparaison numérique.

Si vous utilisez une fonction compliquée pour extraire la partie sur laquelle vous souhaitez faire le tri, il vaut mieux ne pas le faire à l'intérieur de la fonction `sort`. L'extraire d'abord, parce que le BLOC de tri peut être appelé plusieurs fois pour un même élément. Voici par exemple comment récupérer le premier mot après le premier nombre de chaque élément, et ensuite faire le tri sur ces mots sans tenir compte de la casse des caractères.

```
@idx = ();
for (@data) {
 ($item) = /\d+\s*(\S+)/;
 push @idx, uc($item);
}
@sorted = @data[sort { $idx[$a] cmp $idx[$b] } 0 .. $#idx];
```

qui pourrait aussi s'écrire ainsi, en utilisant une astuce qui est connue sous le nom de Transformation Schwartzienne :

```
@sorted = map { $_->[0] }
 sort { $a->[1] cmp $b->[1] }
 map { [$_, uc(/\d+\s*(\S+)/)] } @data;
```

Si vous avez besoin de trier sur plusieurs champs, le paradigme suivant est utile :

```
@sorted = sort { field1($a) <=> field1($b) ||
 field2($a) cmp field2($b) ||
 field3($a) cmp field3($b)
 } @data;
```

Ceci pouvant être aisément combiné avec le calcul préalable des clefs comme détaillé ci-dessus.

Voir l'article *sort* dans la collection "Far More Than You Ever Wanted To Know" sur <http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz> pour plus détails sur cette approche.

Voir aussi ci-dessous la question relative au tri des tables de hachage.

### 21.5.15 Comment manipuler des tableaux de bits ?

Utiliser `pack()` et `unpack()`, ou encore `vec()` et les opérations bit à bit.

Par exemple, ceci impose à `$vec` d'avoir le bit `N` positionné si `$ints[N]` était positionné :

```
$vec = '';
foreach(@ints) { vec($vec, $_, 1) = 1 }
```

Voici comment, avec un vecteur dans `$vec`, amener ces bits dans votre tableau d'entiers `@ints` :

```
sub bitvec_to_list {
 my $vec = shift;
 my @ints;
 # Choisir le meilleur algorithme suivant la densité de bits nuls
 if ($vec =~ tr/\0// / length $vec > 0.95) {
 use integer;
 my $i;
 # Méthode rapide avec surtout des bits nuls
 while($vec =~ /[^\0]/g) {
 $i = -9 + 8 * pos $vec;
```

```

 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 push @ints, $i if vec($vec, ++$i, 1);
 }
} else {
 # Algorithme général rapide
 use integer;
 my $bits = unpack "b*", $vec;
 push @ints, 0 if $bits =~ s/^(\\d)// && $1;
 push @ints, pos $bits while($bits =~ /1/g);
}
return \@ints;
}

```

Cette méthode devient d'autant plus rapide que le vecteur en bits est clairsemé. (Avec la permission de Tim Bunce et Winfried Koenig.)

Vous pouvez raccourcir un peu la boucle grâce à cette suggestions de Benjamin Goldberg :

```

while($vec =~ /[^\0]+/g) {
 push @ints, grep vec($vec, $_, 1), $-[0] * 8 .. $+[0] * 8;
}

```

Ou utiliser le module Bit::Vector du CPAN :

```

$vector = Bit::Vector->new($num_of_bits);
$vector->Index_List_Store(@ints);
@ints = $vector->Index_List_Read();

```

Bit::Vector fournit des méthodes efficaces pour gérer des vecteurs de bits, de petit entiers ou de "big int".

Voici un exemple plus complet d'utilisation de vec() :

```

démo de vec
$vector = "\xff\x0f\xef\xfe";
print "Ilya's string \\xff\\x0f\\xef\\xfe represents the number ",
 unpack("N", $vector), "\n";
$is_set = vec($vector, 23, 1);
print "Its 23rd bit is ", $is_set ? "set" : "clear", ".\n";
pvec($vector);

set_vec(1,1,1);
set_vec(3,1,1);
set_vec(23,1,1);

set_vec(3,1,3);
set_vec(3,2,3);
set_vec(3,4,3);
set_vec(3,4,7);
set_vec(3,8,3);
set_vec(3,8,7);

set_vec(0,32,17);
set_vec(1,32,17);

```

```

sub set_vec {
 my ($offset, $width, $value) = @_;
 my $vector = '';
 vec($vector, $offset, $width) = $value;
 print "offset=$offset width=$width value=$value\n";
 pvec($vector);
}

sub pvec {
 my $vector = shift;
 my $bits = unpack("b*", $vector);
 my $i = 0;
 my $BASE = 8;

 print "vector length in bytes: ", length($vector), "\n";
 @bytes = unpack("A8" x length($vector), $bits);
 print "bits are: @bytes\n\n";
}

```

### 21.5.16 Pourquoi defined() retourne vrai sur des tableaux et hachages vides ?

La petite histoire est que vous ne devriez probablement utiliser defined que sur les scalaires ou les fonctions, et pas sur les agrégats (tableaux et hachages). Voir defined in *perlfunc* dans les versions 5.004 et suivantes de Perl pour plus de détails.

## 21.6 Données : tables de hachage (tableaux associatifs)

### 21.6.1 Comment traiter une table entière ?

Utiliser la fonction each() (voir each in *perlfunc*) si vous n'avez pas besoin de la trier :

```

while (($key, $value) = each %hash) {
 print "$key = $value\n";
}

```

Si vous la souhaitez triée, il vous faudra utiliser foreach() sur le résultat du tri des clefs, comme montré dans une question précédente.

### 21.6.2 Que se passe-t-il si j'ajoute ou j'enlève des clefs d'un hachage pendant que j'itère dessus ?

(contribution de brian d foy)

La réponse simple est « Ne le faites pas ! »

Si vous parcourez la table de hachage via each(), la seule clé que vous pouvez supprimer sans problème est la clé courante. Si vous supprimez ou ajoutez d'autres clés, l'itérateur pourrait oublier certaines clés ou présenter deux fois la même clé puisque perl risque de réarranger la table de hachage. Voir each() dans *perlfunc*.

### 21.6.3 Comment rechercher un élément d'un hachage par sa valeur ?

Créer un hachage inversé :

```

%by_value = reverse %by_key;
$key = $by_value{$value};

```

Ceci n'est pas particulièrement efficace. Il aurait été plus efficace pour l'espace de stockage d'utiliser :

```

while (($key, $value) = each %by_key) {
 $by_value{$value} = $key;
}

```



Si votre hachage pouvait avoir plus d'une valeur identique, les méthodes ci-dessus ne trouveront qu'une seule des clefs associées. Ceci peut être ou ne pas être un problème pour vous. Si cela vous inquiète, vous pouvez toujours inverser le hachage en un hachage de tableaux :

```
while (($key, $value) = each %by_key) {
 push @{$key_list_by_value{$value}}, $key;
}
```

#### 21.6.4 Comment savoir combien d'entrées sont dans un hachage ?

Si vous voulez dire combien de clefs, alors il vous suffit de prendre dans son sens scalaire la fonction keys() :

```
$num_keys = keys %hash;
```

La fonction keys() réinitialise les itérateurs de la table de hachage, ce qui signifie que vous pourriez avoir des résultats étranges si vous l'utilisez pendant l'utilisation sur la même table d'autres opérateurs de table de hachage tels que each().

#### 21.6.5 Comment trier une table de hachage (par valeur ou par clef) ?

(contribution de brian d foy)

Pour trier une table de hachage, commençons par nous intéresser aux clés. Dans l'exemple suivant, nous fournissons la liste des clés à la fonction sort qui les trie ASCIIbétiquement (cet ordre peut-être modifié par vos réglages de locale). Une fois les clés récupérées dans le bon ordre, nous pouvons les utiliser pour afficher un rapport listant ces clés dans l'ordre ASCIIbétique.

```
my @keys = sort { $a cmp $b } keys %hash;

foreach my $key (@keys)
{
 printf "%-20s %6d\n", $key, $hash{$value};
}
```

Nous pouvons utiliser le bloc de sort() d'une manière plus puissante. Au lieu de comparer directement les clés, nous pouvons calculer une valeur à partir de chaque clé et utiliser ces valeurs dans la comparaison.

Par exemple, pour rendre notre ordre d'affichage insensible à la casse, nous utilisons la séquence \L dans une chaîne entre guillemets pour tout transformer en minuscules. Le bloc sort() comparera donc les valeurs des clés en minuscules pour les ordonner.

```
my @keys = sort { "\L$a" cmp "\L$b" } keys %hash;
```

Note : si la transformation des valeurs des clés est coûteuse ou si la table a beaucoup d'éléments, vous pouvez jeter un oeil à la Transformation Schwartzienne pour placer le résultat de la transformation en cache.

Si nous voulons trier notre table de hachage selon l'ordre de ses valeurs, nous utilisons les clés pour y accéder. Nous obtenons toujours une liste de clés mais ordonnée selon les valeurs.

```
my @keys = sort { $hash{$a} <=> $hash{$b} } keys %hash;
```

À partir de là, nous pouvons faire des choses plus complexes. Par exemple, si les valeurs sont identiques, nous pouvons alors utiliser les clés comme ordre secondaire.

```
my @keys = sort {
 $hash{$a} <=> $hash{$b}
 or
 "\L$a" cmp "\L$b"
} keys %hash;
```

### 21.6.6 Comment conserver mes tables de hachage dans l'ordre ?

Vous pouvez regarder dans le module `DB_File` et attacher avec `tie()` en utilisant les liens de hachage `$DB_TREE` tel que documenté dans `DB_File/In Memory Databases`. Le module `Tie::IxHash` du CPAN peut aussi être instructif.

### 21.6.7 Quelle est la différence entre "delete" et "undef" pour des tables de hachage ?

Les table de hachage contiennent des paires de scalaires : le premier est la clef, le second est la valeur. La clef sera toujours convertie en une chaîne alors que la valeur peut être n'importe quelle sorte de scalaire : chaîne, nombre ou référence. Si la clef `$key` est présente dans la table, `exists($hash{$key})` renverra vrai. La valeur associée à une clef donnée peut être `undef`, auquel cas `$hash{$key}` vaudra `undef` tandis que `exists $hash{$key}` retournera vrai. Ceci correspond au fait d'avoir dans la table de hachage la paire (`$key`, `undef`).

Les dessins aident... Voici la table de hachage `%hash` :

```

 clefs valeurs
+-----+-----+
a	3
x	7
d	0
e	2
+-----+-----+

```

La valeur de vérité de quelques conditions est alors :

```

$hash{'a'} est vrai
$hash{'d'} est faux
defined $hash{'d'} est vrai
defined $hash{'a'} est vrai
exists $hash{'a'} est vrai (Perl5 seulement)
grep ($_ eq 'a', keys %hash) est vrai

```

Si vous dites maintenant :

```
undef $ary{'a'}
```

La table devient la suivante :

```

 clefs valeurs
+-----+-----+
a	undef
x	7
d	0
e	2
+-----+-----+

```

Et voici les nouvelles valeurs de vérité de nos conditions (les changements sont en majuscules) :

```

$hash{'a'} est FAUX
$hash{'d'} est faux
defined $hash{'d'} est vrai
defined $hash{'a'} est FAUX
exists $hash{'a'} est vrai (Perl5 seulement)
grep ($_ eq 'a', keys %hash) est vrai

```

Remarquez les deux dernières : on a une valeur `undef`, mais une clef définie !

Maintenant considérez ceci :

```
delete $hash{'a'}
```

La table se lit maintenant :

```

 clefs valeurs
+-----+-----+
x	7
d	0
e	2
+-----+-----+

```

Et nos nouvelles valeurs de vérité sont (avec les changements en majuscules) :

```

$hash{'a'} est faux
$hash{'d'} est faux
defined $hash{'d'} est vrai
defined $hash{'a'} est faux
exists $hash{'a'} est FAUX (Perl5 seulement)
grep ($_ eq 'a', keys %hash) est FAUX

```

Voyez, tout ce qui est lié à cette clé a disparu !

### 21.6.8 Pourquoi mes tables de hachage liées (par tie()) ne font pas la distinction entre exists et defined ?

Cela dépend de l'implémentation de EXISTS() à laquelle est attachée la table de hachage. Par exemple, les tables liées à des fichiers DBM\* ne connaissent pas la notion de valeur undef. Cela signifie donc que exists() et defined() font la même chose sur un fichier DBM\* et que ce qu'elles font ne correspond pas à ce qu'elles feraient sur une table de hachage ordinaire.

### 21.6.9 Comment réinitialiser une opération each() non terminée ?

L'utilisation de keys %hash dans un contexte scalaire renvoie le nombre de clefs d'un hachage et réinitialiser les itérateurs associés à la table de hachage. Vous pouvez en avoir besoin lorsque vous sortez prématurément d'une boucle avec un last afin de réinitialiser l'itérateur de cette boucle lorsque vous voudrez la réutiliser.

### 21.6.10 Comment obtenir l'unicité des clefs de deux hachages ?

Tout d'abord, extraire les clefs des hachages dans des listes, puis résoudre le problème de la "suppression des doublons" tel que décrit plus haut. Par exemple :

```

%dejavu = ();
for $element (keys(%foo), keys(%bar)) {
 $dejavu{$element}++;
}
@uniq = keys %dejavu;

```

Ou plus succinctement :

```
@uniq = keys %{{%foo,%bar}};
```

Ou, pour vraiment économiser de la place :

```

%dejavu = ();
while (defined ($key = each %foo)) {
 $dejavu{$key}++;
}
while (defined ($key = each %bar)) {
 $dejavu{$key}++;
}
@uniq = keys %dejavu;

```

### 21.6.11 Comment enregistrer un tableau multidimensionnel dans un fichier DBM ?

Soit vous transformez vous-mêmes la structure en une chaîne de caractères (casse-tête), soit vous récupérez le module MLDBM (qui utilise Data::Dumper) du CPAN, et vous l'utilisez au-dessus de DB\_File ou GDBM\_File.

### 21.6.12 Comment faire en sorte que mon hachage conserve l'ordre des éléments que j'y mets ?

Utiliser le module Tie::IxHash du CPAN.

```
use Tie::IxHash;
tie my %myhash, 'Tie::IxHash';
for (my $i=0; $i<20; $i++) {
 $myhash{$i} = 2*$i;
}
my @keys = keys %myhash;
@keys = (0,1,2,3,...)
```

### 21.6.13 Pourquoi le passage à un sous-programme d'un élément non défini d'un hachage le crée du même coup ?

Si on dit quelque chose comme :

```
somefunc($hash{"nonesuch key here"});
```

Alors cet élément "s'autoactive" ; c'est-à-dire qu'il se crée tout seul, que l'on y range quelque chose ou pas. Ceci arrive parce que les fonctions reçoivent des scalaires passés par références. Si somefunc() modifie \$\_[0], elle doit être prête à la réécrire dans la version de l'appelant.

Ceci a été réglé à partir de Perl5.004.

Normalement, d'accéder simplement à la valeur d'une clef dans le cas d'une clef inexistante ne produit *pas* une clef présente éternellement. Ceci est différent du comportement d'awk.

### 21.6.14 Comment faire l'équivalent en Perl d'une structure en C, d'une classe/d'un hachage en C++ ou d'un tableau de hachages ou de tableaux ?

Habituellement via une référence de hachage, peut-être comme ceci :

```
$record = {
 NAME => "Jason",
 EMPNO => 132,
 TITLE => "deputy peon",
 AGE => 23,
 SALARY => 37_000,
 PALS => ["Norbert", "Rhys", "Phineas"],
};
```

Les références sont documentées dans *perlref* et le futur *perlrefut*. Des exemples de structures de données complexes sont données dans *perldsc* et *perllol*. Des exemples de structures et de classes orientées objet sont dans *perltoot*.

### 21.6.15 Comment utiliser une référence comme clef d'une table de hachage ?

(contribution de brian d foy)

Les clés des tables de hachage sont des chaînes donc vous ne pouvez pas vraiment utiliser une référence comme clé. Si vous essayez de le faire, perl transforme la référence en une chaîne (par exemple HASH(0xDEADBEEF)). Vous ne pouvez pas retrouver la référence originale à partir de cette chaîne, tout du moins sans fournir vous-même un travail supplémentaire. Souvenez-vous aussi que les clés sont uniques mais deux variables différentes peuvent contenir la même référence (et ces variables peuvent ensuite changer).

La module Tie::RefHash, qui est distribué avec perl, est peut-être ce que vous cherchez. Il gère le travail supplémentaire.

## 21.7 Données : divers

### 21.7.1 Comment manipuler proprement des données binaires ?

Perl est adapté au binaire, donc ceci ne devrait pas être un problème. Par exemple, ceci marche correctement (en supposant les fichiers trouvés) :

```
if ('cat /vmunix' =~ /gzip/) {
 print "Your kernel is GNU-zip enabled!\n";
}
```

Sur les systèmes moins élégants (lire : byzantins), on doit cependant jouer à des jeux éreintants en séparant les fichiers textes ("text") des fichiers binaires ("binary"). Voir `binmode` in *perlfunc* ou *perlopentut*.

Si le problème provient de données ASCII 8-bits, alors voir *perllocale*.

Si vous souhaitez agir sur des caractères multi-octets, alors il y a quelques pièges. Voir la section sur les expressions régulières.

### 21.7.2 Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ?

En supposant que vous ne vous occupiez pas des notations IEEE comme "NaN" ou "Infinity", il vous suffit probablement d'utiliser une expression régulière.

```
if (/^D/) { print "has nondigits\n" }
if (/^\d+$/) { print "is a whole number\n" }
if (/^-?\d+$/) { print "is an integer\n" }
if (/^[+-]?\d+$/) { print "is a +/- integer\n" }
if (/^-?\d+\.\d*$/) { print "is a real number\n" }
if (/^-?(?:\d+(?:\.\d*)?|\.\d+)$/) { print "is a decimal number\n" }
if (/^[+-]?(?:=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/)
 { print "a C float\n" }
```

Il existe aussi quelques modules communs pour faire cela. *Scalar::Util* (distribué avec perl 5.8) donne accès à la fonction interne de perl `looks_like_number` qui permet de déterminer si une valeur ressemble à un nombre. *Data::Types* exporte des fonctions qui valident le type de donnée en utilisant les expressions ci-dessus ainsi que d'autres expressions rationnelles. Il y a aussi *Regexp::Common* qui propose des expressions rationnelles pour reconnaître différents types de nombres. Ces trois modules sont disponibles sur CPAN.

Sur un système POSIX, Perl accepte la fonction `POSIX::strtod`. Sa sémantique est quelque peu malcommode, donc il y a une fonction d'emballage `getnum` pour un usage plus aisé. Cette fonction prend une chaîne et retourne le nombre qu'elle a trouvé, ou `undef` pour une entrée qui n'est pas un nombre à virgule flottante du C. La fonction `is_numeric` est une couverture frontale à `getnum`, au cas où vous voudriez simplement demander « Ceci est-il un nombre à virgule flottante ? »

```
sub getnum {
 use POSIX qw(strtod);
 my $str = shift;
 $str =~ s/^\s+//;
 $str =~ s/\s+$//;
 $! = 0;
 my($num, $unparsed) = strtod($str);
 if (($str eq '') || ($unparsed != 0) || $!) {
 return undef;
 } else {
 return $num;
 }
}

sub is_numeric { defined &getnum($_[0]) }
```

À la place, vous pouvez également regarder le module *String::Scanf* sur CPAN. Le module `POSIX` (qui fait partie de la distribution standard de Perl) propose les fonctions `strtoul` et `strtod` pour convertir des chaînes respectivement en longs et en doubles.

### 21.7.3 Comment conserver des données persistantes entre divers appels de programme ?

Pour des application spécifiques, on peut utiliser l'un des modules DBM. Voir `AnyDBM_File`. Plus généralement, vous devriez consulter les modules `FreezeThaw` ou `Storable` du CPAN. Depuis Perl 5.8, `Storable` fait partie de la distribution standard. Voici un exemple utilisant les fonctions `store` et `retrieve` de `Storable` :

```
use Storable;
store(\%hash, "filename");

later on...
$href = retrieve("filename"); # par référence
%hash = %{ retrieve("filename") }; # accès direct au hachage
```

### 21.7.4 Comment afficher ou copier une structure de données récursive ?

Le module `Data::Dumper` du CPAN (ou en standard depuis Perl 5.005) est agréable pour afficher les structures de données. Le module `Storable` sur CPAN (ou en standard depuis Perl 5.8) fournit une fonction appelée `dclone` qui copie récursivement ses arguments.

```
use Storable qw(dclone);
$r2 = dclone($r1);
```

Où `$r1` peut être une référence à tout type de structure de données. Il sera copié en profondeur. Puisque `dclone` prend et renvoie des références, vous devez ajouter de la ponctuation si c'est un hachage de tableaux que vous désirez copier.

```
%newhash = %{ dclone(\%oldhash) };
```

### 21.7.5 Comment définir des méthodes pour toutes les classes ou tous les objets ?

Utiliser la classe `UNIVERSAL` (voir `UNIVERSAL`).

### 21.7.6 Comment vérifier la somme de contrôle d'une carte de crédit ?

Récupérer le module `Business::CreditCard` du CPAN.

### 21.7.7 Comment compacter des tableaux de nombres à virgule flottante simples ou doubles pour le code XS ?

Le code `kgbpack.c` dans le module `PGPLOT` du CPAN fait pile cela. Si vous faites beaucoup de traitement de nombres à virgule flottante, vous pouvez envisager d'utiliser à la place le module `PDL` du CPAN – il rend la chose plus facile.

## 21.8 AUTEUR ET COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen, Nathan Torkington et autres auteurs suscités. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code précisant l'origine serait de bonne courtoisie mais n'est pas indispensable.

## 21.9 TRADUCTION

La traduction française est distribuée avec les même droits que sa version originale (voir ci-dessus).

### **21.9.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **21.9.2 Traducteur**

Traduction initiale : Guillaume van der Rest <gvdr@dcmr.polytechnique.fr>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit <paul.gaborit@enstimac.fr>

### **21.9.3 Relecture**

Régis Julié <Regis.Julie@cetelem.fr>, Gérard Delafond

# Chapitre 22

## perlfaq5

Fichiers et formats

### 22.1 DESCRIPTION

Cette section traite de E/S (Entrées et Sorties) et autres éléments connexes : descripteurs de fichiers, vidage de tampons, formats d'écriture et pieds de page.

#### 22.1.1 Comment vider ou désactiver les tampons en sortie ? Pourquoi m'en soucier ?

Perl ne propose pas vraiment de sorties sans tampon (sauf en passant par `syswrite(OUT, $char, 1)`) mais plutôt une sorte de tampon par commande où une écriture réelle est effectuée après chaque commande de sortie.

La bibliothèque standard d'E/S du C (`stdio`) accumule normalement les caractères envoyés aux divers périphériques dans des tampons dans le but d'éviter d'effectuer un appel système pour chaque octet. Dans la plupart des implémentations de `stdio`, le type d'accumulation en sortie et la taille des tampons varient suivant le périphérique utilisé. Les fonctions Perl `print()` et `write()` utilisent normalement ces tampons alors que `syswrite()` passe outre.

Si vous souhaitez que vos sorties soient envoyées immédiatement lorsque vous effectuez un `print()` ou un `write()` (par exemple, dans le cas de certains protocoles réseau), il vous faudra activer le mode d'écriture systématique (`autoflush`) des tampons attachés au descripteur de fichier. Ce mode est représenté par la variable Perl `$|` et si elle contient une valeur vraie, Perl videra le tampon du descripteur de fichier après chaque appel à `print()` ou `write()`. Une modification de `$|` modifie la gestion des tampons du descripteur de fichier sélectionné par défaut. Vous pouvez choisir ce descripteur de fichier via un appel à la fonction `select()` avec un seul argument (voir `perlvar|$|` et `select` in *perlfunc*).

Utilisez `select()` pour choisir le bon descripteur de fichier puis positionnez les variables de pilotage.

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Ou, de façon plus idiomatique, en une seule instruction :

```
select((select(OUTPUT_HANDLE), $| = 1)[0]);

$| = 1, select $_ for select OUTPUT_HANDLE;
```

Certains modules proposent un accès orienté objet aux descripteurs de fichiers et à leur paramètres (ils seront sans doute un peu lourd si vous ne leur demandez que cela). Par exemple `IO::Handle` :

```
use IO::Handle;
open(DEV, ">/dev/printer"); # à quoi ça sert ?
DEV->autoflush(1);
```

Ou `IO::Socket` :

```
use IO::Socket; # une sorte de tube ?
my $sock = IO::Socket::INET->new('www.example.com:80');

$sock->autoflush();
```



### 22.1.2 Comment changer une ligne, effacer une ligne, insérer une ligne au milieu ou ajouter une ligne en tête d'un fichier ?

Utiliser le module `Tie::File` qui fait partie de la distribution standard depuis Perl 5.8.0.

### 22.1.3 Comment déterminer le nombre de lignes d'un fichier ?

Un moyen assez efficace est de compter les caractères de fin de ligne dans le fichier. Le programme suivant utilise une propriété de `tr///`, décrite dans *perlop*. Si votre fichier de texte ne se termine pas par un caractère de fin de ligne, alors ce n'est pas vraiment un fichier de texte correct, et ce programme vous surprendra en indiquant une ligne de moins.

```
$lines = 0;
open(FILE, $filename) or die "Can't open '$filename': $!";
while (sysread FILE, $buffer, 4096) {
 $lines += ($buffer =~ tr/\n//);
}
close FILE;
```

On supposera qu'il n'y a aucune traduction parasite des caractères de fin de ligne à déplorer.

### 22.1.4 Comment utiliser l'option `-i` de Perl depuis l'intérieur d'un programme ?

L'option `-i` modifie la valeur de la variable Perl `$_I` qui modifie le comportement de `<>` ; voir *perlrun* pour plus de détails. En modifiant directement les bonnes variables, vous pouvez obtenir le même comportement dans votre programme. Par exemple :

```
...
{
 local($_I, @ARGV) = ('.orig', glob("*.c"));
 while (<>) {
 if ($. == 1) {
 print "Cette ligne sera en tete de chaque fichier\n";
 }
 s/\b(p)earl\b/${1}erl/i; # Correction en tenant
 # compte de la casse
 print;
 close ARGV if eof; # Réinitialise $.
 }
}
$_I et @ARGV reprennent ici leur ancienne valeur
```

Ce bloc de code modifie tous les fichier `.c` du répertoire courant en créant une copie de sauvegarde de chaque fichier original dans un fichier `.c.orig`.

### 22.1.5 Comment puis-je copier un fichier ?

(contribution de brian d foy)

Utilisez le module `File::Copy`. Il vient avec Perl et sait faire de vraies copies entre différents systèmes de fichiers, tout cela de manière portable.

```
use File::Copy;

copy($original, $new_copy) or die "Échec de la copie : $!";
```

Si vous ne pouvez pas utiliser `File::Copy`, vous devrez faire le travail vous-même : ouvrez le fichier original, ouvrez le fichier de destination puis écrivez dans le fichier de destination au fur et à mesure que vous lisez le fichier original.

### 22.1.6 Comment créer un fichier temporaire ?

Si vous n'avez pas besoin de connaître le nom du fichier temporaire, vous pouvez utiliser `open()` en passant `undef` à la place d'un nom de fichier. Le fonction `open()` créera alors un fichier temporaire anonyme.

```
open my $tmp, '+>', undef or die $!;
```

Sinon vous pouvez utiliser le module `File::Temp`.

```
use File::Temp qw/ tempfile tempdir /;

$dir = tempdir(CLEANUP => 1);
($fh, $filename) = tempfile(DIR => $dir);

ou si vous n'avez pas besoin du nom du fichier

$fh = tempfile(DIR => $dir);
```

Le module `File::Temp` est un module standard depuis Perl 5.6.1. Si vous ne disposez pas d'une version moderne de Perl, utilisez la méthode de classe `new_tmpfile` du module `IO::File` pour obtenir un descripteur de fichier ouvert en lecture écriture. À utiliser si le nom dudit fichier importe peu.

```
use IO::File;
$fh = IO::File->new_tmpfile()
 or die "Impossible de créer un fichier temporaire : $!";
```

Si vous tenez absolument à tout faire vous-même, utilisez l'ID du processus et/ou la valeur du compteur de temps. Si vous avez besoin de plusieurs fichiers temporaires, ayez recours à un compteur :

```
BEGIN {
 use Fcntl;
 my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMPDIR} || $ENV{TEMP};
 my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
 sub temp_file {
 local *FH;
 my $count = 0;
 until (defined(fileno(FH)) || $count++ > 100) {
 $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
 # O_EXCL est indispensable pour la sécurité.
 sysopen(FH, $base_name, O_WRONLY|O_EXCL|O_CREAT);
 }
 if (defined(fileno(FH)))
 return (*FH, $base_name);
 } else {
 return ();
 }
 }
}
```

### 22.1.7 Comment manipuler un fichier avec des enregistrements de longueur fixe ?

Le plus efficace est d'utiliser `pack()` et `unpack()`. C'est plus rapide que d'utiliser `substr()` sur de très nombreuses chaînes. C'est plus lent pour seulement quelques unes.

Voici un morceau de code démontrant comment découper et ensuite réassembler des lignes formatées selon un schéma donné, ici la sortie du programme `ps`, version Berkeley :

```

exemple de ligne à traiter
15158 p5 T 0:00 perl /home/ram/bin/scripts/now-what
my $PS_T = 'A6 A4 A7 A5 A*';
open my $ps, '|-', 'ps';
print scalar <$ps>;
my @fields = qw(pid tt stat time command);
while (<$ps>) {
 my %process;
 @process{@fields} = unpack($PS_T, $_);
 for my $field (@fields) {
 print "$field: <$process{$field}>\n";
 }
 print 'line=', pack($PS_T, @process{@fields}), "\n";
}

```

Nous avons utilisé une tranche de table de hachage afin de gérer facilement les champs de chaque ligne. Le stockage des clés dans un tableau permet de les utiliser facilement toutes ensemble et de leur appliquer des boucles. Cela évite aussi de polluer le programme avec des variables globales ou d'utiliser des références symboliques.

### 22.1.8 Comment rendre un descripteur de fichier local à une routine ? Comment passer des descripteurs à d'autres routines ? Comment construire un tableau de descripteurs ?

Depuis perl 5.6, `open()` autovivifie les descripteurs de fichier ou de répertoire en tant que référence si vous lui passez une variable scalaire non définie. Vous pouvez passer ces références à d'autres routines comme n'importe quel autre scalaire et les utiliser à la place des noms de descripteur.

```

open my $fh, $file_name;

open local $fh, $file_name;

print $fh "Hello World!\n";

process_file($fh);

```

Avant perl 5.6, il fallait jongler avec différentes formes idiomatiques liées aux types universels (`typeglob`). Vous les rencontrerez dans d'anciens codes.

```

open FILE, "> $filename";
process_typeglob(*FILE);
process_reference(*FILE);

sub process_typeglob { local *FH = shift; print FH "Typeglob!" }
sub process_reference { local $fh = shift; print $fh "Reference!" }

```

Si vous voulez créer plusieurs descripteurs anonymes, vous devriez jeter un oeil aux modules `Symbol` et `IO::Handle`.

### 22.1.9 Comment utiliser un descripteur de fichier indirectement ?

Un descripteur de fichier indirect s'utilise par l'intermédiaire d'une variable placée là où, normalement, le langage s'attend à trouver un descripteur de fichier. On obtient une telle variable ainsi :

```

$fh = SOME_FH; # un mot brut est mal-aimé de 'strict subs'
$fh = "SOME_FH"; # mal-aimé de 'strict refs'; même package seulement
$fh = *SOME_FH; # type universel
$fh = *SOME_FH; # référence sur un type universel (bénissable)
$fh = *SOME_FH{IO}; # IO::Handle béni du type universel *SOME_FH

```

Ou en utilisant la méthode `new` de l'un des modules `IO::*` pour créer un descripteur anonyme, l'affecter dans une variable scalaire, puis l'utiliser ensuite comme si c'était un descripteur de fichier normal.

```
use IO::Handle; # 5.004 ou mieux
$fh = IO::Handle->new();
```

Vous pouvez alors utiliser ces objets comme un descripteur de fichier normal. Partout où Perl s'attend à trouver un descripteur de fichier, un descripteur indirect peut être substitué. Ce descripteur indirect est tout simplement une variable scalaire contenant un descripteur de fichier. Des fonctions comme `print`, `open`, `seek` ou l'opérateur diamant `<FH>` acceptent soit un descripteur de fichier sous forme de nom soit une variable scalaire contenant un descripteur :

```
($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
print $ofh "Type it: ";
$got = <$ifh>;
print $efh "What was that: $got";
```

Quand on veut passer un descripteur de fichier à une fonction, il y a deux manières d'écrire la routine :

```
sub accept_fh {
 my $fh = shift;
 print $fh "Sending to indirect filehandle\n";
}
```

Ou il est possible de localiser un type universel (typeglob) et d'utiliser le nom de descripteur ainsi obtenu directement :

```
sub accept_fh {
 local *FH = shift;
 print FH "Sending to localized filehandle\n";
}
```

Ces deux styles marchent aussi bien avec des objets, des types universels ou des descripteurs de fichiers réels. (Ils pourraient aussi se contenter de chaînes simples, dans certains cas, mais c'est plutôt risqué.)

```
accept_fh(*STDOUT);
accept_fh($handle);
```

Dans les exemples ci-dessus, nous avons affecté le descripteur de fichier à une variable scalaire avant de l'utiliser. La raison est que seules de simples variables scalaires, par opposition à des expressions ou des notations indicées dans des tableaux normaux ou associatifs, peuvent être ainsi utilisées avec des fonctions natives comme `print`, `printf`, ou l'opérateur diamant. Les exemples suivants sont invalides et ne passeront pas la phase de compilation :

```
@fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: "; # INVALIDE
$got = <$fd[0]> # INVALIDE
print $fd[2] "What was that: $got"; # INVALIDE
```

Avec `print` et `printf`, on peut s'en sortir avec un bloc contenant une expression à la place du descripteur de fichier normalement attendu :

```
print { $fd[1] } "funny stuff\n";
printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
On obtient dans $fd[1] : Pity the poor deadbeef.
```

Ce bloc est un bloc ordinaire, semblable à tout autre, donc on peut y placer des expressions plus complexes. Ceci envoie le message vers une destination parmi deux :

```
$ok = -x "/bin/cat";
print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
print { $fd[1+ ($ok || 0)] } "cat stat $ok\n";
```

Cette façon de traiter `print` et `printf` comme si c'étaient des appels à des méthodes objets ne fonctionne pas avec l'opérateur diamant. Et ce parce que c'est vraiment un opérateur et pas seulement une fonction avec un argument spécial, non délimité par une virgule. En supposant que l'on ait stocké divers types universels dans une structure, comme montré ci-avant, on peut utiliser la fonction native `readline` pour lire un enregistrement comme le ferait `<>`. Avec l'initialisation montrée ci-dessus pour `@fd`, cela marcherait, mais seulement parce que `readline()` demande un type universel. Cela ne marchera pas avec des objets ou des chaînes, ce qui pourrait bien être un de ces bugs non encore corrigés.

```
$got = readline($fd[0]);
```

Notons ici que cet exotisme des descripteurs indirects ne dépend pas du fait qu'ils peuvent prendre la forme de chaînes, types universels, objets, ou autres. C'est simplement dû à la syntaxe des opérateurs fondamentaux. Jouer à l'orienté objet ne serait d'aucune aide ici.

### 22.1.10 Comment mettre en place un pied-de-page avec write() ?

Il n'y a pas de méthode native pour accomplir cela, mais *perform* indique une ou deux techniques qui permettent aux programmeurs intrépides de s'en sortir.

### 22.1.11 Comment rediriger un write() dans une chaîne ?

Voir Accéder aux formats de l'intérieur in *perform* pour un exemple de fonction swrite().

### 22.1.12 Comment afficher mes nombres avec des virgules pour délimiter les milliers ?

(contribution de brian d foy et de Benjamin Goldberg)

Vous pouvez utiliser le module *Number::Format* pour délimiter vos nombres. Ce module tient compte des informations fournies par les 'locale' pour ceux qui voudraient utiliser un espace comme délimiteur (ou n'importe quel autre caractère).

La routine suivante ajoute des virgules dans un nombre :

```
sub commify {
 local $_ = shift;
 1 while s/^([-+]? \d+) (\d{3}) /$1,$2/;
 return $_;
}
```

Cette expression rationnelle de Benjamin Goldberg ajoute des virgules aux nombres :

```
s/(^[-+]? \d+?(?=(?>(?:\d{3})+)(?! \d)) | \G \d{3}(?=\d)) /$1,/g;
```

Elle est plus lisible avec des commentaires :

```
s/(
 ^[-+]? # début d'un nombre.
 \d+? # les premiers chiffres avant une virgule
 (?= # suivis par,
 # (sans faire partie de ce qui est reconnu)
 (?>(?:\d{3})+) # un ou plusieurs groupes de 3 chiffres
 (?! \d) # un multiple *exact* et non x * 3 + 1 ou autre.
)
 | # ou :
 \G \d{3} # le dernier groupe avec 3 chiffres
 (?=\d) # mais sans aucun chiffre ensuite.
)/$1,/xg;
```

### 22.1.13 Comment traduire les tildes (~) dans un nom de fichier ?

Utiliser l'opérateur <> (appelé glob()), tel que décrit dans *perlfunc*. Les versions anciennes de Perl nécessitaient la présence d'un shell qui comprenne les tildes. Les versions récentes de Perl gèrent cette fonction nativement. Le module *Glob::KGlob* (disponible sur CPAN) implémente une fonctionnalité de glob plus portable.

En Perl, on peut utiliser ceci directement :

```
$filename =~ s{
 ^ ~ # cherche le tilde en tête
 (# sauvegarde dans $1 :
 [^/] # tout caractère sauf un slash
 * # et ce 0 ou plusieurs fois (0 pour mon propre login)
)
}{
 $1
 ? (getpwnam($1))[7]
 : ($ENV{HOME} || $ENV{LOGDIR})
}ex;
```

### 22.1.14 Pourquoi les fichiers que j'ouvre en lecture-écriture se voient-ils effacés ?

Parce que vous faites quelque chose du genre indiqué ci-après, qui tronque d'abord le fichier et *seulement ensuite* donne un accès en lecture-écriture :

```
open(FH, "+> /path/name"); # MAUVAIS (en général)
```

Il faudrait faire comme ceci, ce qui échouera si le fichier n'existe pas déjà.

```
open(FH, "+< /path/name"); # ouvert pour mise à jour
```

L'utilisation de ">" crée ou met toujours à zéro. L'utilisation de "<" ne le fait jamais. Le "+" n'y change rien.

Voici différents exemples d'ouverture. Tous ceux qui utilisent `sysopen()` supposent que l'on a déjà fait :

```
use Fcntl;
```

Pour ouvrir un fichier en lecture :

```
open(FH, "< $path") || die $!;
sysopen(FH, $path, O_RDONLY) || die $!;
```

Pour ouvrir un fichier en écriture, en le créant si c'est un nouveau fichier ou en le tronquant s'il est préexistant :

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666) || die $!;
```

Pour ouvrir un fichier en écriture, en créant un fichier qui n'existe pas déjà :

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666) || die $!;
```

Pour ouvrir un fichier avec ajout en fin, en le créant si nécessaire :

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

Pour ouvrir un fichier existant avec ajout en fin :

```
sysopen(FH, $path, O_WRONLY|O_APPEND) || die $!;
```

Pour ouvrir un fichier existant en mode de mise à jour :

```
open(FH, "+< $path") || die $!;
sysopen(FH, $path, O_RDWR) || die $!;
```

Pour ouvrir un fichier en mode de mise à jour, avec création si besoin :

```
sysopen(FH, $path, O_RDWR|O_CREAT) || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666) || die $!;
```

Pour ouvrir en mise à jour un fichier qui n'existe pas déjà :

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT) || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666) || die $!;
```

Enfin, pour ouvrir un fichier sans bloquer, avec création éventuelle :

```
sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT)
 or die "can't open /tmp/somefile: $!";
```

Attention : ni la création, ni la destruction de fichier n'est garantie être atomique à travers NFS. C'est-à-dire que deux processus pourraient simultanément arriver à créer ou à effacer le même fichier sans erreur. En d'autres termes, `O_EXCL` n'est pas aussi exclusif que ce que l'on pourrait penser de prime abord.

Voir aussi la nouvelle page de documentation *perlopentut* si vous en disposez (à partir de la version 5.6).

### 22.1.15 Pourquoi <\*> donne de temps en temps l'erreur "Argument list too long" ?

L'opérateur <> est utilisé pour faire appel à glob() (cf. ci-dessus). Dans les versions de Perl précédant la v5.6.0, l'opérateur glob() interne lance csh(1) pour effectuer ladite complétion, mais csh ne peut pas traiter plus de 127 éléments et retourne donc le message d'erreur `Argument list too long`. Ceux qui ont installé tcsh à la place de csh n'auront pas ce problème, mais les utilisateurs de leur code pourront en être surpris.

Pour contourner cela, soit vous mettez Perl à jour vers une version 5.6.0 ou supérieur, soit vous faites votre complétion vous-même avec readdir() et des motifs, soit vous utilisez un module comme File::KGlob, qui n'a pas recours au shell pour faire cette complétion.

### 22.1.16 Y a-t-il une fuite / un bug avec glob() ?

De par son implémentation sur certains systèmes d'exploitation, l'utilisation de la fonction glob(), directement ou sous sa forme <> dans un contexte scalaire, peut provoquer une fuite mémoire ou un comportement non-prédictible. Il est donc préférable de n'utiliser glob() que dans un contexte de liste.

### 22.1.17 Comment ouvrir un fichier dont le nom commence par ">" ou se termine par des espaces ?

(contribution de Brian McCauley)

La fonction Perl open(), appelée dans sa forme spéciale à deux arguments, ignore les espaces en fin de nom de fichier et déduit le mode d'ouverture du fichier en se basant sur la présence de certains caractères au début du nom (ou d'un "|" en fin). Dans les anciennes versions de Perl, c'était le seul moyen d'utiliser la fonction open() si bien qu'on retrouve cette forme dans de nombreux codes ou livres anciens.

À moins d'avoir des raisons particulières d'utiliser cette forme d'appel à deux arguments, vous devriez utiliser l'appel à trois arguments qui ne traite aucun caractère comme étant spécial dans le nom du fichier.

```
open FILE, "<", " fichier "; # le nom du fichier est " fichier "
open FILE, ">", ">fichier"; # le nom du fichier est ">fichier"
```

### 22.1.18 Comment renommer un fichier de façon sûre ?

Si votre système d'exploitation fournit un programme mv(1) correct ou équivalent moral, ceci fonctionnera :

```
rename($old, $new) or system("mv", $old, $new);
```

Pour être plus portable, il peut être intéressant d'utiliser le module File::Copy. On copie simplement le fichier sur le nouveau nom (en vérifiant bien les codes de retour de chaque fonction), puis on efface l'ancien nom. En revanche, cela n'a pas tout à fait la même sémantique que le véritable rename() qui, lui, préservera des méta-informations comme les droits, les dates diverses et autres informations liées au fichier.

Les versions récentes de File::Copy fournissent une fonction move().

### 22.1.19 Comment verrouiller un fichier ?

La fonction flock() fournie par Perl (cf. *perlfunc* pour plus de détails) appelle flock(2) si elle est disponible, sinon fcntl(2) (pour les versions de perl supérieure à 5.004) ou finalement lockf(3) si aucun des deux appels systèmes précédents n'est disponible. Sur certains systèmes, une forme native de verrouillage peut même être utilisée. Voici quelques avertissements relatifs à l'utilisation du flock() de Perl :

1. La fonction produit une erreur fatale si aucun des trois appels (ou proches équivalents) n'existe.
2. lockf(3) ne fournit pas de verrouillage partagé et impose que le descripteur de fichier soit ouvert au minimum en mode écriture (et donc aussi en mode ajout ou en mode lecture/écriture).

3. Certaines versions de flock() ne peuvent pas verrouiller de fichiers à travers un réseau (par exemple via NFS). En conséquence, vous devriez forcer Perl à utiliser fcntl(2) lors de sa compilation. Mais même ceci n'est pas sûr. Voir à ce sujet flock dans *perlfunc* et le fichier *INSTALL* dans la distribution source pour savoir comment procéder.

Deux sémantiques potentielles de flock peu évidentes mais traditionnelles sont qu'il attend indéfiniment jusqu'à obtenir le verrouillage et qu'il verrouille *simplement pour information*. De tels verrouillages discrétionnaires sont plus flexibles, mais offrent des garanties moindres. Ceci signifie que les fichiers verrouillés avec flock() peuvent être modifiés par des programmes qui eux n'utilisent pas flock(). Les voitures qui respectent les feux de signalisation s'entendent bien entre elles, mais pas avec les voitures qui grillent les feux rouges. Voir *perlport*, la documentation spécifique à votre portage, ou vos pages de manuel locales spécifiques à votre système pour plus de détails. Il est conseillé de choisir un comportement traditionnel si vous écrivez des programmes portables (mais si ce n'est pas le cas, vous êtes absolument libre d'écrire selon les idiosyncrasies de votre propre système (parfois appelées des "caractéristiques"). L'adhérence aveugle aux soucis de portabilité ne devrait pas vous empêcher de faire votre boulot).

Pour plus d'informations sur le verrouillage de fichiers, voir aussi Verrouillage de fichier in *perlopentut* si vous en disposez (à partir de la version 5.6).

### 22.1.20 Pourquoi ne pas faire simplement open(FH, ">file.lock") ?

Un bout de code À NE PAS UTILISER est :

```
sleep(3) while -e "file.lock"; # MERCI de NE PAS UTILISER
open(LCK, "> file.lock"); # ce code FOIREUX
```

C'est un cas classique de conflit d'accès (race condition) : on fait en deux temps quelque chose qui devrait être réalisé en une seule opération. C'est pourquoi les microprocesseurs fournissent une instruction atomique appelée test-and-set. En théorie, ceci devrait fonctionner :

```
sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
 or die "can't open file.lock: $!";
```

sauf que, lamentablement, la création (ou l'effacement) n'est pas atomique à travers NFS, donc cela ne marche pas (du moins, pas tout le temps) à travers le réseau. De multiples schémas utilisant link() ont été suggérés, mais ils ont tous tendance à mettre en jeu une boucle d'attente active, ce qui est tout autant indésirable.

### 22.1.21 Je ne comprends toujours pas le verrouillage. Je veux seulement incrémenter un compteur dans un fichier. Comment faire ?

Ne vous a-t-on jamais expliqué que les compteurs d'accès aux pages web étaient inutiles ? Ils ne comptent pas vraiment le nombre d'accès, ils sont une perte de temps et ils ne servent qu'à gonfler la vanité de l'auteur. Il vaudrait mieux choisir un nombre au hasard. Ce serait plus réaliste.

Quoiqu'il en soit, voici ce que vous pouvez faire si vous ne pouvez vraiment pas vous en empêcher :

```
use Fcntl ':flock';
sysopen(FH, "numfile", O_RDWR|O_CREAT) or die "can't open numfile: $!";
flock(FH, LOCK_EX) or die "can't flock numfile: $!";
$num = <FH> || 0;
seek(FH, 0, 0) or die "can't rewind numfile: $!";
truncate(FH, 0) or die "can't truncate numfile: $!";
(print FH $num+1, "\n") or die "can't write numfile: $!";
close FH or die "can't close numfile: $!";
```

Voici un bien meilleur compteur d'accès aux pages web :

```
$hits = int((time() - 850_000_000) / rand(1_000));
```

À défaut de la valeur du compteur lui-même, ce code pourrait impressionner vos amis... :-)



### 22.1.22 Je souhaite juste ajouter un peu de texte à la fin d'un fichier. Dois-je tout de même utiliser le verrouillage ?

Si vous utilisez un système qui implémente correctement `flock()` et si vous utilisez l'exemple de code d'ajout proposé par "perldoc -f flock", tout devrait bien se passer même si votre système ne gère pas bien le mode d'ajout (si un tel système existe). Donc, si vous vous limitez aux systèmes qui implémentent `flock()` (et ce n'est pas vraiment une limitation), tout ira bien.

Si vous êtes sûr que les systèmes visés implémentent correctement le mode d'ajout (c.-à-d. pas Win32) alors vous pouvez même omettre le `seek()` dans le code évoqué ci-dessus.

Si vous êtes sûr d'écrire du code pour un système d'exploitation et un système de fichiers qui implémentent correctement l'ajout (par exemple, un système de fichier local sur un Unix moderne) et si vous laissez le fichier en mode tampon par bloc et si vous la taille de ce que vous écrivez n'excède pas la taille du tampon entre chaque écriture réelle manuelle, alors vous avez la garantie que l'écriture du tampon à la fin du fichier se fera de manière atomique sans risque de mélange avec d'autres écritures. Vous pouvez aussi utiliser la fonction `syswrite()` qui n'est qu'un habillage de la fonction système `write(2)`.

Il existe encore un risque infime qu'un signal interrompe l'appel système `write()` avant sa terminaison. Certains implémentations de STDIO peuvent aussi, rarement, effectuer plusieurs appels à `write()` alors que le tampon était bien vide au préalable. Il y a certains systèmes où la probabilité que cela arrive est proche de zéro.

### 22.1.23 Comment modifier un fichier binaire directement ?

Si vous souhaitez juste modifier un binaire, un code simple, comme ci-dessous, fonctionne bien.

```
perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

En revanche, si vous avez des enregistrements de taille fixe, alors vous pourriez faire plutôt comme ceci :

```
$RECSIZE = 220; # taille en octets de l'enregistrement
$recno = 37; # numéro d'enregistrement à modifier
open(FH, "+<somewhere") || die "can't update somewhere: $!";
seek(FH, $recno * $RECSIZE, 0);
read(FH, $record, $RECSIZE) == $RECSIZE || die "can't read record $recno: $!";
modifie l'enregistrement
seek(FH, -$RECSIZE, 1);
print FH $record;
close FH;
```

Le verrouillage et le traitement des erreurs sont laissés en exercice au lecteur. Ne les oubliez pas, ou vous vous en mordrez les doigts.

### 22.1.24 Comment récupérer la date d'un fichier en perl ?

Si vous voulez récupérer la dernière date à laquelle le fichier a été lu, écrit ou a vu ses méta-informations (propriétaire, etc.) changées, utilisez les opérations de test sur fichier `-M`, `-A` ou `-C`, documentées dans *perlfunc*. Elles récupèrent l'âge du fichier (mesuré par rapport à l'heure de démarrage du programme) exprimée en fraction de jours. Selon la plateforme, certaines de ces dates ne sont pas disponibles. Pour récupérer l'âge "brut" en secondes depuis l'origine des temps (du système), il faut utiliser la fonction `stat()`, puis utiliser `localtime()`, `gmtime()` ou `POSIX::strftime()` pour convertir ce nombre dans un format lisible par un humain.

Voici un exemple :

```
$write_secs = (stat($file))[9];
printf "file %s updated at %s\n", $file,
 scalar localtime($write_secs);
```

Si vous préférez quelque chose de plus lisible, utilisez le module `File::stat` (qui fait partie de la distribution standard depuis la version 5.004) :

```
gestion des erreurs laissée en exercice au lecteur.
use File::stat;
use Time::localtime;
$date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

L'approche `POSIX::strftime()` a le bénéfice d'être, en théorie, indépendante de la localisation courante. Voir *perllocale* pour plus de détails.

### 22.1.25 Comment modifier la date d'un fichier en perl ?

Utilisez la fonction `utime()` documentée dans `utime` in *perlfunc*. À titre d'exemple, voici un petit programme qui applique récupère les dates de dernier accès et de dernière modification de son premier argument et les applique à tous les autres :

```
if (@ARGV < 2) {
 die "usage: cptimes timestamp_file other_files ...\n";
}
$timestamp = shift;
($atime, $mtime) = (stat($timestamp))[8,9];
utime $atime, $mtime, @ARGV;
```

Comme d'habitude, le traitement des erreurs est laissé en exercice au lecteur.

La documentation de la fonction `utime()` donne aussi un exemple qui a le même effet que la fonction `touch(1)` sur les fichiers *existants*.

Certains systèmes de fichiers limitent la précision de stockage de ces dates. Par exemple, les systèmes de fichiers FAT et HPFS ne peuvent pas descendre en dessous de deux secondes de précision. Ce sont des limitations de ces systèmes de fichiers et non de la fonction `utime()`.

### 22.1.26 Comment écrire dans plusieurs fichiers simultanément ?

Pour connecter un descripteur de fichier à plusieurs descripteurs en sortie, vous pouvez utiliser les modules `IO::Tee` ou `Tie::FileHandle::Multiplex`.

Pour une utilisation unique, on peut recourir à :

```
for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

### 22.1.27 Comment lire le contenu d'un fichier d'un seul coup ?

Vous pouvez utiliser le module `File::Slurp` pour faire cela :

```
use File::Slurp;

$all_of_it = read_file($filename);
tout le fichier dans un scalaire
@all_lines = read_file($filename);
une ligne du fichier par élément
```

L'approche habituelle en Perl pour traiter toutes les lignes d'un fichier est de le faire ligne par ligne :

```
open (INPUT, $file) || die "can't open $file: $!";
while (<INPUT>) {
 chomp;
 # traiter la ligne qui est dans $_
}
close(INPUT) || die "can't close $file: $!";
```

Ceci est nettement plus efficace que de lire le fichier tout entier en mémoire en tant que tableau de lignes puis de le traiter élément par élément, ce qui est souvent (sinon presque toujours) la mauvaise approche. Chaque fois que vous voyez quelqu'un faire ceci :

```
@lines = <INPUT>;
```

vous devriez réfléchir longuement et profondément à la raison justifiant que tout soit chargé en même temps. Ce n'est tout simplement pas une solution extensible. Vous pourriez aussi trouver plus amusant d'utiliser le module `Tie::File` ou les liens `$DB_RECNO` du module standard `DB_File`, qui vous permettent de lier un tableau à un fichier tel que l'accès à un élément du tableau accède en vérité à la ligne correspondante dans le fichier.

Vous pouvez lire la totalité du fichier dans un scalaire :

```
{
 local(*INPUT, $/);
 open (INPUT, $file) || die "can't open $file: $!";
 $var = <INPUT>;
}
```

Ce code donne temporairement la valeur indéfinie à votre séparateur d'enregistrements et ferme automatiquement le fichier à la sortie du bloc. Si le fichier est déjà ouvert, utilisez juste ceci :

```
$var = do { local $/; <INPUT> };
```

Pour des fichiers ordinaires, vous pouvez aussi utiliser la fonction `read` :

```
read(INPUT, $var, -s INPUT);
```

Le troisième argument détermine le nombre d'octets contenus dans le filehandle `INPUT` pour tous les lire dans la variable `$var`.

### 22.1.28 Comment lire un fichier paragraphe par paragraphe ?

Utilisez la variable `$/` (voir *perlvar* pour plus de détails). Vous pouvez la positionner soit à `""` pour éliminer les paragraphes vides ("`abc\n\n\ndef`", par exemple, sera traité comme deux paragraphes et non trois), soit à `"\n\n"` pour accepter les paragraphes vides.

Notez qu'une ligne blanche ne doit pas contenir de blancs. Ainsi, "`fred\n \nstuff\n\n`" est seul un paragraphe alors que "`fred\n\nstuff\n\n`" en fait deux.

### 22.1.29 Comment lire un seul caractère d'un fichier ? Et du clavier ?

Vous pouvez utiliser la fonction native `getc()` sur la plupart des descripteurs de fichier, mais elle ne marchera pas (facilement) sur un terminal. Pour `STDIN`, utilisez soit le module `Term::ReadKey` disponible sur CPAN, soit l'exemple fourni dans `getc` in *perlfunc*.

Si votre système supporte POSIX (portable operating system programming interface), vous pouvez utiliser le code suivant qui, vous l'aurez noté, supprime aussi l'écho pendant le traitement.

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
 my $got;
 print "gimme: ";
 $got = getone();
 print "--> $got\n";
}
exit;

BEGIN {
 use POSIX qw(:termios_h);

 my ($term, $oterm, $echo, $noecho, $fd_stdin);
```

```

$fd_stdin = fileno(STDIN);

$term = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm = $term->getlflag();

$echo = ECHO | ECHOK | ICANON;
$noecho = $oterm & ~$echo;

sub cbreak {
 $term->setlflag($noecho);
 $term->setcc(VTIME, 1);
 $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
 $term->setlflag($oterm);
 $term->setcc(VTIME, 0);
 $term->setattr($fd_stdin, TCSANOW);
}

sub getone {
 my $key = '';
 cbreak();
 sysread(STDIN, $key, 1);
 cooked();
 return $key;
}

}

END { cooked() }

```

Le module `Term::ReadKey` de CPAN est sans doute plus facile à utiliser. Les versions récentes incluent aussi un support pour les systèmes non portables.

```

use Term::ReadKey;
open(TTY, "</dev/tty");
print "Gimme a char: ";
ReadMode "raw";
$key = ReadKey 0, *TTY;
ReadMode "normal";
printf "\nYou said %s, char number %03d\n",
 $key, ord $key;

```

### 22.1.30 Comment savoir si un caractère est disponible sur un descripteur de fichier ?

La première chose à faire, c'est de se procurer le module `Term::ReadKey` depuis CPAN. Comme nous le mentionnions plus haut, il contient même désormais un support limité pour les systèmes non portables (comprendre : non ouverts, fermés, propriétaires, non POSIX, non Unix, etc.).

Vous devriez aussi lire la Foire Aux Questions de `comp.unix.*` pour ce genre de chose : la réponse est sensiblement identique. C'est très dépendant du système d'exploitation utilisé. Voici une solution qui marche sur les systèmes BSD :

```

sub key_ready {
 my($rin, $nfd);
 vec($rin, fileno(STDIN), 1) = 1;
 return $nfd = select($rin, undef, undef, 0);
}

```

Si vous désirez savoir combien de caractères sont en attente, regardez du côté de `ioctl()` et de `FIONREAD`. L'outil `h2ph` qui est fourni avec Perl essaie de convertir les fichiers d'inclusion du C en code Perl, qui peut alors être utilisé via `require`. `FIONREAD` se retrouve défini comme une fonction dans le fichier `sys/ioctl.ph` :

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size) or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

Si `h2ph` n'est pas installé ou s'il ne fonctionne pas pour vous, il est possible d'utiliser directement `grep` sur les fichiers d'en-tête :

```
% grep FIONREAD /usr/include/*/*
/usr/include/asm/ioctls.h:#define FIONREAD 0x541B
```

Ou écrivez un petit programme C, en utilisant l'éditeur des champions :

```
% cat > fionread.c
#include <sys/ioctl.h>
main() {
 printf("%#08x\n", FIONREAD);
}
^D
% cc -o fionread fionread.c
% ./fionread
0x4004667f
```

Puis, utilisez directement cette valeur, en laissant le problème de portage comme exercice à votre successeur.

```
$FIONREAD = 0x4004667f; # XXX: depend du système d'exploitation

$size = pack("L", 0);
ioctl(FH, $FIONREAD, $size) or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

`FIONREAD` impose un descripteur connecté à un canal (stream), ce qui signifie qu'il fonctionne bien avec les prises (sockets), les tubes (pipes) et les terminaux (tty) mais *pas* avec les fichiers.

### 22.1.31 Comment écrire un `tail -f` en perl ?

Essayez d'abord :

```
seek(GWFILE, 0, 1);
```

La ligne `seek(GWFILE, 0, 1)` ne change pas la position courante, mais elle efface toute indication de fin de fichier sur le descripteur, de sorte que le prochain `<GWFILE>` conduira Perl à essayer de nouveau de lire quelque chose.

Si cela ne fonctionne pas (cela demande certaines propriétés à votre implémentation stdio), alors vous pouvez essayer quelque chose comme :

```
for (;;) {
 for ($curpos = tell(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
 # cherche des trucs, et mets-les quelque part
 }
 # attendre un peu
 seek(GWFILE, $curpos, 0); # retourne où nous en étions
}
```

Si cela ne marche pas non plus, regardez du côté du module POSIX. POSIX définit la fonction `clearerr()` qui peut ôter la condition de fin de fichier sur le descripteur. La méthode : lire jusqu'à obtenir une fin de fichier, `clearerr()`, lire la suite. Nettoyer, rincer, et ainsi de suite.

Il existe aussi un module `File::Tail` sur le CPAN.

### 22.1.32 Comment faire un dup() sur un descripteur en Perl ?

Voir `open` in *perlfunc* pour obtenir divers moyens d'appeler `open()` qui pourraient vous convenir. Par exemple :

```
open(LOG, ">>/tmp/logfile");
open(STDERR, ">&LOG");
```

Ou même avec des descripteurs numériques :

```
$fd = $ENV{MHCONTEXTFD};
open(MHCONTEXT, "<&=$fd"); # comme fdopen(3S)
```

Noter que "<&STDIN" donne un clone, mais que "<&=STDIN" donne un alias. Cela veut dire que si vous fermez un descripteur possédant des alias, ceux-ci deviennent indisponibles. C'est faux pour un clone.

Comme d'habitude, le traitement d'erreur est laissé en exercice au lecteur.

### 22.1.33 Comment fermer un descripteur connu par son numéro ?

Cela n'est que très rarement nécessaire puisque la fonction `close()` de Perl n'est censée être utilisée que pour des choses ouvertes par Perl, même si c'est un clone (dup) de descripteur numérique comme `MHCONTEXT` ci-dessus. Mais si c'est absolument nécessaire, vous pouvez essayer :

```
require 'sys/syscall.ph';
$rc = syscall(&SYS_close, $fd + 0); # doit forcer une valeur numérique
die "can't sysclose $fd: $!" unless $rc == -1;
```

Ou bien utilisez juste la caractéristique `fdopen(3S)` de `open()` :

```
{
 local *F;
 open F, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
 close F;
}
```

### 22.1.34 Pourquoi "C:\temp\foo" n'indique pas un fichier DOS ? Et même "C:\temp\foo.exe" ne marche pas ?

Ouille ! Vous venez de mettre une tabulation et un formfeed dans le nom du fichier. Rappelez-vous qu'à l'intérieur des guillemets ("`\tel quel`"), le backslash est un caractère d'échappement. La liste complète de telles séquences est dans Opérateurs apostrophe et type apostrophe in *perlop*. Evidemment, vous n'avez pas de fichier appelé "`c:(tab)emp(formfeed)oo`" ou "`c:(tab)emp(formfeed)oo.exe`" sur votre système de fichiers DOS originel.

Utilisez soit des guillemets simples (apostrophes) pour délimiter vos chaînes, ou (mieux) utilisez des slashes. Toutes les versions de DOS et de Windows venant après MS-DOS 2.0 traitent / et \ de la même façon dans les noms de fichier, donc autant utiliser une forme compatible avec Perl – ainsi qu'avec le shell POSIX, ANSI C et C++, awk, Tcl, Java, ou Python, pour n'en mentionner que quelques autres. Les chemins POSIX sont aussi plus portables.

### 22.1.35 Pourquoi glob("\*.\*") ne donne-t-il pas tous les fichiers ?

Parce que, même sur les portages non-Unix, la fonction `glob()` de Perl suit la sémantique normale de complétion d'Unix. Il faut utiliser `glob("*")` pour obtenir tous les fichiers (non cachés). Cela contribue à rendre `glob()` portable y compris sur les systèmes ancestraux. Votre portage peut aussi inclure des fonctions de globalisation propriétaires. Regardez sa documentation pour plus de détails.

### 22.1.36 Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi -i écrit-il dans des fichiers protégés ? N'est-ce pas un bug de Perl ?

Ce sujet est traité de façon complète et fastidieuse dans un article de la collection "Far More Than You Ever Wanted To Know" disponible sur <http://www.perl.com/CPAN/doc/FMTEYEWTK/file-dir-perms> .

En résumé, apprenez comment fonctionne votre système de fichiers. Les permissions sur un fichier indiquent seulement ce qui peut arriver aux données dudit fichier. Les permissions sur le répertoire indiquent ce qui peut survenir à la liste des fichiers contenus dans ce répertoire. Effacer un fichier revient à ôter son nom de la liste du répertoire (donc l'opération est régie par les permissions sur le répertoire, pas sur le fichier). Si vous essayez d'écrire dans le fichier alors les permissions du fichiers sont prises en compte pour déterminer si vous en avez le droit.

### 22.1.37 Comment sélectionner une ligne au hasard dans un fichier ?

Voici un algorithme tiré du Camel Book :

```
srand;
rand($.) < 1 && ($line = $_) while <>;
```

Il a un énorme avantage en espace par rapport à la solution consistant à tout lire en mémoire. Une preuve que cette méthode est correcte se trouve dans *The Art of Computer Programming*, Volume 2, Section 3.4.2, par Donald E. Knuth.

Vous pouvez utiliser le module `File::Random` qui fournit une fonction reposant sur cette méthode.

```
use File::Random qw/random_line/;
my $line = random_line($filename);
```

Un autre moyen passe par le module `Tie::File` qui donne accès à tout un fichier comme si c'était un tableau. Il suffit alors de choisir au hasard un élément du tableau.

### 22.1.38 Pourquoi obtient-on des espaces étranges lorsqu'on affiche un tableau de lignes ?

Le fait de dire :

```
print "@lines\n";
```

joint les éléments de `@lines` avec un espace entre eux. Si `@lines` contient ("little", "fluffy", "clouds") alors l'instruction précédente affiche :

```
little fluffy clouds
```

mais si chaque élément de `@lines` est une ligne de texte, terminée par une fin de ligne, ("little\n", "fluffy\n", "clouds\n"), alors elle affiche :

```
little
fluffy
clouds
```

Si votre tableau contient déjà des lignes, affichez les simplement :

```
print @lines;
```

## 22.2 AUTHOR AND COPYRIGHT (on Original English Version)

Copyright (c) 1997-1999 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

Copyright (c) 1997-1999 Tom Christiansen et Nathan Torkington. Tous droits réservés.

Cette documentation est libre. Vous pouvez la redistribuer ou la modifier sous les mêmes conditions que Perl lui-même.

Indépendamment de sa distribution, tous les exemples de code sont placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code et ses dérivés dans vos propres programmes, réalisés soit pour le plaisir, soit par profit, comme bon vous semble. Une simple mention dans le code créditant cette FAQ serait une marque de politesse mais n'est pas obligatoire.

## 22.3 TRADUCTION

### 22.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 22.3.2 Traducteur

Raphaël Manfredi <Raphael\_Manfredi@grenoble.hp.com>.

Roland Trique <roland.trique@uhb.fr>, Paul Gaborit *paul.gaborit @ enstimac.fr* (mise à jour).

### 22.3.3 Relecture

Régis Julie <Régis.Julie@Cetelem.fr>, Gérard Delafond.



# Chapitre 23

## perlfaq6

Expressions rationnelles

### 23.1 DESCRIPTION

Cette partie de la FAQ est incroyablement courte car les autres parties sont parsemées de réponses concernant les expressions rationnelles. Par exemple, décoder une URL ou vérifier si quelque chose est un nombre ou non relève du domaine des expressions rationnelles, mais ces réponses se trouvent ailleurs que dans ce document (dans *perlfaq9* « Comment décoder ou créer ces %-encodings sur le web ? » et dans *perlfaq4* « Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ? »).

#### 23.1.1 Comment utiliser les expressions rationnelles sans créer du code illisible et difficile à maintenir ?

Trois méthodes peuvent rendre les expressions rationnelles faciles à maintenir et compréhensibles.

##### Commentaires en dehors de l'expression rationnelle

Décrivez ce que vous faites et comment vous le faites en utilisant la façon habituelle de commenter en Perl.

```
transforme la ligne en le premier mot, le signe deux points
et le nombre de caractères du reste de la ligne
s/^(\\w+)(.*)/ lc($1) . ":" . length($2) /meg;
```

##### Commentaires dans l'expression rationnelle

En utilisant le modificateur `/x`, les espaces seront ignorés dans le motif de l'expression rationnelle (sauf dans une classe de caractères) et vous pourrez ainsi utiliser les commentaires habituels dedans. Comme vous pouvez l'imaginer, espaces et commentaires aident pas mal.

Un `/x` vous permet de transformer ceci :

```
s{<(?:[>'"]*|'.*'|'.*?')+>}gs;
```

en :

```
s{ < # signe inférieur
 (?: # parenthèse ouvrante ne créant pas de référence
 [^>'"] * # 0 ou plus de caract. qui ne sont ni >, ni ' ni "
 | # ou
 ".*?" # une partie entre guillemets (reconnaissance min)
 | # ou
 '.*?' # une partie entre apostrophes (reconnaissance min)
) + # tout cela se produisant une ou plusieurs fois
 > # signe supérieur
}gsx; # remplacé par rien, c.-à-d. supprimé
```

Ce n'est pas encore aussi clair que de la prose, mais c'est très utile pour décrire le sens de chaque partie du motif.

**Différents délimiteurs**

Bien qu'habituellement délimités par le caractère /, les motifs peuvent en fait être délimités par quasiment n'importe quel caractère. *perlre* l'explique. Par exemple, `s///` ci-dessus utilise des accolades comme délimiteurs. Sélectionner un autre délimiteur permet d'éviter de le quoter à l'intérieur du motif :

```
s/\usr\local\usr\share/g; # mauvais choix de délimiteur
s#/usr/local#/usr/share#g; # meilleur
```

**23.1.2 J'ai des problèmes pour faire une reconnaissance sur plusieurs lignes. Qu'est-ce qui ne va pas ?**

Ou vous n'avez en fait pas plus d'une ligne dans la chaîne où vous faites la recherche (cas le plus probable), ou vous n'appliquez pas le bon modificateur à votre motif (cas possible).

Il y a plusieurs manières d'avoir plusieurs lignes dans une chaîne. Si vous voulez l'avoir automatiquement en lisant l'entrée, vous initialiserez `$/` (probablement à "" pour des paragraphes ou `undef` pour le fichier entier) ce qui vous permettra de lire plus d'une ligne à la fois.

Lire *perlre* vous aidera à décider lequel des modificateurs `/s` ou `/m` (ou les deux) vous utiliserez : `/s` autorise le point à reconnaître le caractère fin de ligne alors que `/m` permet à l'accent circonflexe et au dollar de reconnaître tous les débuts et fins de ligne, pas seulement le début et la fin de la chaîne. Vous pouvez utiliser cela pour être sûr que vous avez bien plusieurs lignes dans votre chaîne.

Par exemple, ce programme détecte les mots répétés, même dans des lignes différentes (mais pas dans plusieurs paragraphes). Pour cet exemple, nous n'avons pas besoin de `/s` car nous n'utilisons pas le point dans l'expression rationnelle qui doit enjamber les fins de ligne. Nous n'avons pas besoin non plus de `/m` car nous ne voulons pas que le circonflexe ou le dollar fasse une reconnaissance d'un début ou d'une fin d'une nouvelle ligne. Mais il est impératif que `$/` ait une autre valeur que la valeur par défaut, ou alors nous n'aurons pas plusieurs lignes d'un coup à se mettre sous la dent.

```
$/ = ''; # lis au moins un paragraphe entier,
 # pas qu'une seule ligne
while (<>) {
 while (/\b([\w'-]+)(\s+\1)+\b/gi) { # les mots commencent par
 # des caractères alphanumériques
 print "$1 est répété dans le paragraphe $.\n";
 }
}
```

Voilà le code qui trouve les phrases commençant avec "From " (qui devrait être transformés par la plupart des logiciels de courrier électronique) :

```
$/ = ''; # lis au moins un paragraphe entier, pas qu'une seule ligne
while (<>) {
 while (/^From /gm) { # /m fait que ^ reconnaisse
 # tous les débuts de ligne
 print "from de départ dans le paragraphe $.\n";
 }
}
```

Voilà le code qui trouve tout ce qu'il y a entre START et END dans un paragraphe :

```
undef $/; # lis le fichier entier, pas seulement qu'une ligne
 # ou un paragraphe
while (<>) {
 while (/START(.*?)END/sgm) { # /s fait que . enjambe les lignes
 print "$1\n";
 }
}
```

### 23.1.3 Comment extraire des lignes entre deux motifs qui sont chacun sur des lignes différentes ?

Vous pouvez utiliser l'opérateur quelque peu exotique `..` de Perl (documenté dans *perlop*):

```
perl -ne 'print if /START/ .. /END/' fichier1 fichier2 ...
```

Si vous voulez du texte et non des lignes, vous pouvez utiliser

```
perl -0777 -ne 'print "$1\n" while /START(.*)END/gs' fichier1 fichier2 ...
```

Mais si vous voulez des occurrences imbriquées de `START` à l'intérieur de `END`, vous tombez sur le problème décrit, dans cette section, sur la reconnaissance de texte bien équilibré.

Ici un autre exemple d'utilisation de `..` :

```
while (<>) {
 $in_header = 1 .. /^$/;
 $in_body = /^$/ .. eof();
 # maintenant choisissez entre eux
} continue {
 reset if eof(); # fixe $.
}
```

### 23.1.4 J'ai mis une expression rationnelle dans `$/` mais cela ne marche pas. Qu'est-ce qui est faux ?

Jusqu'à Perl 5.8 inclus, `$/` doit être une chaîne. Cela pourrait changer en 5.10 mais n'y comptez pas trop tout de même. En attendant, vous pouvez vous baser sur les exemples ci-dessous si vous en avez réellement besoin.

Si vous disposez de `File::Stream`, c'est très facile.

```
use File::Stream;
my $stream = File::Stream->new(
 $filehandle,
 separator => qr/\s*,\s*/,
);

print "$_\n" while <$stream>;
```

Si vous ne disposez pas de `File::Stream`, il vous faudra en faire un peu plus.

Vous pouvez utiliser la forme à quatre argument de `sysread` pour remplir peu à peu un tampon. Après chaque ajout au tampon, vous vérifiez si vous avez une ligne entière (en utilisant votre expression rationnelle).

```
local $_ = "";
while(sysread FH, $_, 8192, length) {
 while(s/^(?.*)your_pattern/) {
 my $record = $1;
 # votre traitement ici.
 }
}
```

Vous pouvez faire la même chose avec un `foreach` testant une expression rationnelle utilisant le modificateur `c` et l'ancrage `\G` si le risque d'avoir tout votre fichier en mémoire à la fin ne vous dérange pas.

```
local $_ = "";
while(sysread FH, $_, 8192, length) {
 foreach my $record (m/\G(?s).*)your_pattern/gc) {
 # votre traitement ici.
 }
 substr($_, 0, pos) = "" if pos;
}
```

### 23.1.5 Comment faire une substitution indépendante de la casse de la partie gauche mais préservant cette casse dans la partie droite ?

Voici une adorable solution perlienne par Larry Rosler. Elle exploite les propriétés du xor bit à bit sur les chaînes ASCII.

```

$_ = "this is a TESt case";

$old = 'test';
$new = 'success';

s{(\Q$old\E)}
{ uc $new | (uc $1 ^ $1) .
 (uc(substr $1, -1) ^ substr $1, -1) x
 (length($new) - length $1)
}egi;

print;

```

Et la voici sous la forme d'un sous-programme, selon le modèle ci-dessus :

```

sub preserve_case($$) {
 my ($old, $new) = @_;
 my $mask = uc $old ^ $old;

 uc $new | $mask .
 substr($mask, -1) x (length($new) - length($old))
}

$a = "this is a TESt case";
$a =~ s/(test)/preserve_case($1, "success")/egi;
print "$a\n";

```

Ceci affiche :

```
this is a SUcCESS case
```

Jeff Pinyan propose une autre manière de faire qui préserve la casse du mot de remplacement s'il s'avère plus long que l'original :

```

sub preserve_case {
 my ($from, $to) = @_;
 my ($lf, $lt) = map length, @_;

 if ($lt < $lf) { $from = substr $from, 0, $lt }
 else { $from .= substr $to, $lf }

 return uc $to | ($from ^ uc $from);
}

```

Cela transformera la phrase en "this is a SUcCess case."

Rien que pour montrer que les programmeurs C peuvent écrire du C dans tous les langages de programmation, si vous préférez une solution plus proche du style de C, le script suivant fait en sorte que la substitution a la même casse, lettre par lettre, que l'original (il s'avère aussi tourner environ 240 % plus lentement que la version perlienne). Si la substitution a plus de caractères que la chaîne substituée, la casse du dernier caractère est utilisée pour le reste de la substitution.

```

L'original est de Nathan Torkington, mis en forme par Jeffrey Friedl
#
sub preserve_case($$)
{
 my ($old, $new) = @_;
 my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
 my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
 my ($len) = $oldlen < $newlen ? $oldlen : $newlen;

```

```

for ($i = 0; $i < $len; $i++) {
 if ($c = substr($old, $i, 1), $c =~ /[W\d_]/) {
 $state = 0;
 } elsif (lc $c eq $c) {
 substr($new, $i, 1) = lc(substr($new, $i, 1));
 $state = 1;
 } else {
 substr($new, $i, 1) = uc(substr($new, $i, 1));
 $state = 2;
 }
}
on se retrouve avec ce qui reste de new
(quand new est plus grand que old)
if ($newlen > $oldlen) {
 if ($state == 1) {
 substr($new, $oldlen) = lc(substr($new, $oldlen));
 } elsif ($state == 2) {
 substr($new, $oldlen) = uc(substr($new, $oldlen));
 }
}
return $new;
}

```

### 23.1.6 Comment faire pour que `\w` reconnaisse les caractères nationaux ?

Placez `use locale;` dans votre script. La classe de caractères `\w` est définie à partir du locale courant. Voir *perllocale* pour les détails.

### 23.1.7 Comment reconnaître une version locale-équivalente de `/[a-zA-Z]/` ?

Vous pouvez utiliser la classe de caractères POSIX `/[[:alpha:]]/` qui est documentée dans *perlre*.

Quel que soit le locale courant, les caractères alphabétiques sont tous les caractères reconnus par `\w` sauf les chiffres et le caractère de soulignement. Exprimé sous la forme d'une expression rationnelle ça ressemble à `/[^\W\d_] /`. Son complémentaire, les caractères non-alphabétiques, est tout `\W` plus les chiffres et le caractère de soulignement, donc `/[\W\d_] /`.

### 23.1.8 Comment protéger une variable pour l'utiliser dans une expression rationnelle ?

L'analyseur syntaxique de Perl remplacera par leur valeur les occurrences de `$variable` et `@variable` dans les expressions rationnelles à moins que le délimiteur ne soit une apostrophe. Rappelez-vous aussi que la partie droite d'une `s///` substitution est considérée comme une chaîne entre guillemets (voir *perlop* pour plus de détails). Rappelez-vous encore que n'importe quel caractère spécial d'expression rationnelle agira à moins que vous ne précédiez la substitution avec `\Q`. Voici un exemple :

```

$chaine = "Placido P. Octopus";
$regex = "P.";

$chaine =~ s/$regex/Polyp/;
$chaine est maintenant "Polypacido P. Octopus"

```

Dans les expressions rationnelles, le `.` est un caractère spécial qui reconnaît n'importe quel caractère, et donc l'expression rationnelle `P.` reconnaît le `P` du début la chaîne originale.

Pour échapper à la signification spéciale du `.`, nous utilisons `\Q` :

```

$chaine = "Placido P. Octopus";
$regex = "P.";

$chaine =~ s/\Q$regex/Polyp/;
$chaine est maintenant "Placido Polyp Octopus"

```

L'utilisation de `\Q` fait que le `.` de l'expression rationnelle est traité comme un caractère normal, et donc `P.` reconnaît maintenant un `P` suivi d'un point.

### 23.1.9 À quoi sert vraiment /o ?

L'utilisation d'une variable dans une opération de reconnaissance d'expression rationnelle force une réévaluation (et peut-être une recompilation) de l'expression rationnelle à chaque fois qu'elle est appliquée. Le modificateur /o verrouille l'expression rationnelle la première fois qu'elle est utilisée. C'est toujours ce qui se passe avec une expression rationnelle constante, et en fait, le motif est compilé dans le format interne en même temps que le programme entier l'est.

Utiliser /o est peu pertinent à moins que le remplacement de variable ne soit utilisé dans le motif, et si cela est, le moteur d'expression rationnelle ne prendra pas en compte les modifications de la variable ultérieures à la *toute première* évaluation.

/o est souvent utilisé pour gagner en efficacité en ne faisant pas les évaluations nécessaires quand vous savez que cela ne pose pas de problème (car vous savez que les variables ne changeront pas), ou plus rarement, quand vous ne voulez pas que l'expression rationnelle remarque qu'elles changent.

Par exemple, voici un programme "paragrep" :

```
$/ = ''; # mode paragraphe
$pat = shift;
while (<>) {
 print if /$pat/o;
}
```

### 23.1.10 Comment utiliser une expression rationnelle pour enlever les commentaires de type C d'un fichier ?

Bien que cela puisse se faire, c'est plus compliqué que vous ne pensez. Par exemple, cette simple ligne

```
perl -0777 -pe 's{/*.*?*/}{}gs' foo.c
```

marchera dans beaucoup de cas mais pas tous. Vous voyez, c'est un peu simplet pour certains types de programmes C, en particulier, ceux où des chaînes protégées sont des commentaires. Pour cela, vous avez besoin de quelque chose de ce genre, créé par Jeffrey Friedl, modifié ultérieurement par Fred Curtis :

```
$/ = undef;
$_ = <>;
s#/*[\^*]**+([\^/*][\^*]**+)*|("(\.|\[^\\"\\])*"|'(\.|\[^\\"\\])*'|\.[^\'"\\"\\]*)#defined $2 ? $2 : ""#gse
print;
```

Cela pourrait, évidemment, être écrit plus lisiblement avec le modificateur /x en ajoutant des espaces et des commentaires. Le voici étendu, une gentillesse de Fred Curtis.

```
s{
 \/* ## Début d'un commentaire /* ... */
 [\^*]**+ ## Non-* suivie par 1 ou plusieurs *
 (
 [\^/*][\^*]**+
)* ## 0 ou plusieurs choses ne commençant pas par /
 ## mais finissent par '*'
 / ## Fin d'un commentaire /* ... */

 | ## OU diverses choses qui ne sont pas des commentaires

 (
 " ## Début d'une chaîne " ... "
 (
 \\. ## Caractère échappé
 | ## OU
 [\^"\\"\\] ## Non "\
)*
 " ## Fin d'une chaîne " ... "
```

```

| ## OU
' ## Début d'une chaîne ' ... '
(
 \\. ## Caractère échappé
| ## OU
 [^\.\] ## Non '\
)*
' ## Fin d'une chaîne ' ... '

| ## OU
. ## Tout autre caractère
[^\.'"\] * ## Caractères ne débutant pas un commentaire,
 ## une chaîne ou un échappement
)
}{{defined $2 ? $2 : ""}}gxse;

```

Une légère modification retire aussi les commentaires C++ :

```
s#/* [^*]* *+ ([^/*] [^*]* *+)* /|// [^\n]* | ("(\. | [^\"])*" | '(\. | [^']*)*' | [^\.'"\] *)#defined $2 ? $
```

### 23.1.11 Est-ce possible d'utiliser les expressions rationnelles de Perl pour reconnaître du texte bien équilibré ?

Auparavant, les expressions rationnelles de Perl ne pouvaient pas reconnaître du texte bien équilibré. Dans les versions récentes de perl depuis la 5.6.1, des fonctionnalités expérimentales ont été ajoutées afin de rendre possible ces reconnaissances. La documentation de la construction (`??{ }`) dans *perlre* montre des exemples de reconnaissances de parenthèses bien équilibrées. Assurez-vous de prendre en compte les avertissements présents dans cette documentation avant d'utiliser ces fonctionnalités.

Le CPAN propose de nombreux modules utiles pour reconnaître des textes dépendant de leur contexte. Damian Conway fournit quelques expressions pratiques dans `Regexp::Common`. Le module `Text::Balanced` fournit une solution générale à ce problème.

XML et HTML font parties des usages courants de la reconnaissance de textes équilibrés. Il existe plusieurs modules qui répondent à ce besoin. Parmi eux `HTML::Parser` et `XML::Parser` mais il y en a bien d'autres.

Une sous-routine élaborée (pour du 7-bit ASCII seulement) pour extraire de simples caractères se correspondant et peut-être imbriqués, comme ' et ', { et }, ou ( et ) peut être trouvée sur [http://www.cpan.org/authors/id/TOMC/scripts/pull\\_quotes.gz](http://www.cpan.org/authors/id/TOMC/scripts/pull_quotes.gz).

Le module `C::Scan` du CPAN contient de tels sous-programmes pour son usage interne, mais elles ne sont pas documentées.

### 23.1.12 Que veut dire "les expressions rationnelles sont gourmandes" ? Comment puis-je le contourner ?

La plupart des gens pensent que les expressions rationnelles gourmandes reconnaissent autant qu'elles peuvent. Techniquement parlant, c'est en réalité les quantificateurs (`?`, `*`, `+`, `{}`) qui sont gourmands plutôt que le motif entier ; Perl préfère les gourmandises locales et les gratifications immédiates à une glotonnerie globale. Pour avoir les versions non gourmandes des mêmes quantificateurs, utilisez (`??`, `*?`, `++`, `{}`).

Un exemple:

```

$s1 = $s2 = "J'ai très très froid";
$s1 =~ s/tr.*s //; # J'ai froid
$s2 =~ s/tr.*?s //; # J'ai très froid

```

Notez que la seconde substitution arrête la reconnaissance dès qu'un "s" est rencontré. Le quantificateur `*?` dit effectivement au moteur des expressions rationnelles de trouver une reconnaissance aussi vite que possible et de passer le contrôle à la suite, comme de se refiler une patate chaude.

### 23.1.13 Comment examiner chaque mot dans chaque ligne ?

Utilisez la fonction `split` :

```
while (<>) {
 foreach $word (split) {
 # faire quelque chose avec $word ici
 }
}
```

Notez que ce ne sont pas vraiment des mots dans le sens mots français ; ce sont juste des suites de caractères différents des caractères d'espace.

Pour travailler seulement avec des séquences alphanumériques (caractère de soulignement inclu), vous pourriez envisager

```
while (<>) {
 foreach $word (m/(\w+)/g) {
 # faire quelque chose avec $word ici
 }
}
```

### 23.1.14 Comment afficher un rapport sur les fréquences de mots ou de lignes ?

Pour faire cela, vous avez à sortir chaque mot du flux d'entrée. Nous appellerons mot une suite de caractères alphabétiques, de tirets et d'apostrophes plutôt qu'une suite de tout caractère sauf espace comme vue dans la question précédente :

```
while (<>) {
 while (/(\b[^\W\d][\w'-]+\b)/g) { # on rate "'mouton'"
 $seen{$1}++;
 }
}
while (($word, $count) = each %seen) {
 print "$count $word\n";
}
```

Si vous voulez faire la même chose avec les lignes, vous n'avez pas besoin d'une expression rationnelle :

```
while (<>) {
 $seen{$_}++;
}
while (($line, $count) = each %seen) {
 print "$count $line";
}
```

Si vous voulez que le résultat soit trié, regardez dans *perlfaq4* : « Comment trier une table de hachage (par valeur ou par clef) ? ».

### 23.1.15 Comment faire une reconnaissance approximative ?

Regardez le module `String::Approx` disponible sur CPAN.

### 23.1.16 Comment reconnaître efficacement plusieurs expressions rationnelles en même temps ?

(contribution de brian d foy)

Ne demandez pas à Perl de recompiler une expression rationnelle à chaque utilisation. Dans l'exemple suivant, perl doit compiler l'expression rationnelle à chaque passage dans la boucle `foreach()` puisqu'il n'a aucun moyen de savoir ce que contient `$motif`.



```
@motifs = qw(foo bar baz);

LINE: while(<>)
{
 foreach $motif (@motifs)
 {
 print if /\b$motif\b/i;
 next LINE;
 }
}
```

L'opérateur `qr//` est apparu dans la version 5.005 de perl. Il compile une expression rationnelle sans l'appliquer. Lorsque vous utilisez la version précompilée de l'expression rationnelle, perl a moins de travail à faire. Dans l'exemple suivant, on introduit un `map()` pour transformer chaque motif en sa version précompilé. Le reste du script est identique mais plus rapide.

```
@motifs = map { qr/\b$_\b/i } qw(foo bar baz);

LINE: while(<>)
{
 foreach $motif (@motifs)
 {
 print if /$motif/;
 next LINE;
 }
}
```

Dans certains cas, vous pouvez même reconnaître plusieurs motifs à l'intérieur d'une même expression rationnelle. Faites tout de même attention aux situations amenant à des retours arrière.

```
$regex = join '|', qw(foo bar baz);

LINE: while(<>)
{
 print if /\b(?:$regex)\b/i;
}
```

Pour plus de détails concernant l'efficacité des expressions rationnelles, lisez « Mastering Regular Expressions » par Jeffrey Freidl. Il explique le fonctionnement du moteur d'expressions rationnelles et pourquoi certains motifs sont étonnamment inefficaces. Une fois que vous aurez bien compris comment perl utilise les expressions rationnelles, vous pourrez les optimiser finement dans toutes les situations.

### 23.1.17 Pourquoi les recherches de limite de mot avec `\b` ne marchent pas pour moi ?

(contribution de brian d foy)

Assurez-vous de bien comprendre ce que `\b` représente : c'est la frontière entre un caractère de mot, `\w`, et quelque chose qui n'est pas un caractère de mot. Ce quelque chose qui n'est pas un caractère de mot peut être un `\W` mais aussi le début ou la fin de la chaîne.

Ce n'est pas la frontière entre un caractère d'espace et un caractère autre et ce n'est pas ce qui sépare les mots d'une phrase.

En terme d'expression rationnelle, une frontière de mot (`\b`) est une « assertion de longueur nulle », ce qui signifie qu'elle ne représente pas un caractère de la chaîne mais une condition à une position donnée.

Dans l'expression rationnelle `\bPerl\b/`, il doit y avoir une frontière de mot avant le "P" et après le "l". À partir du moment où autre chose qu'un caractère de mot précède le "P" et suit le "l", le motif est reconnu. Les chaînes suivantes sont reconnues par `\bPerl\b/`.

```
"Perl" # pas de caractère de mot avant "P" ou après "l"
"Perl " # idem (l'espace n'est pas un caractère de mot)
"'Perl'" # l'apostrophe n'est pas un caractère de mot
"Perl's" # pas de caractère de mot avant "P",
 # un caractère non-mot après "l"
```

Les chaînes suivantes ne sont pas reconnues par `/\bPerl\b/`.

```
"Perl_" # _ est un caractère de mot !
"Perler" # pas de caractère de mot avant "P", mais bien après "l"
```

L'usage de `\b` ne se limite pas aux mots. Vous pouvez chercher des caractères non-mot entourés de caractères de mot. Les chaînes suivantes sont reconnues par le motif `/\b'\b/`.

```
"don't" # l'apostrophe est entourée par "n" et "t"
"qep'a'" # l'apostrophe est entourée par "p" et "a"
```

La chaîne suivante n'est pas reconnue par `/\b'\b/`.

```
"foo'" # aucun caractère de mot après l'apostrophe
```

Vous pouvez aussi utiliser le complémentaire de `\b`, `\B`, pour spécifier qu'il ne doit pas y avoir de frontière de mot.

Dans le motif `/\Bam\B/`, il doit y avoir un caractère de mot avant le "a" et après le "m". Ces chaînes sont reconnues par `/\Bam\B/`.

```
"llama" # "am" est entouré par des caractères de mot
"Samuel" # idem
```

Les chaînes suivantes ne sont pas reconnues par `/\Bam\B/`.

```
"Sam" # pas de frontière de mot avant "a", mais après "m"
"I am Sam" # "am" entourés par des caractères non-mot
```

### 23.1.18 Pourquoi l'utilisation de `$ &`, `$ '`, or `$ '` ralentit tant mon programme ?

(contribution de Anno Siegel)

Une fois que Perl sait que vous avez besoin d'une de ces variables quelque part, il les calcule pour chaque reconnaissance de motif. Cela signifie qu'à chaque reconnaissance de motif, la chaîne entière est recopiée, une partie dans `$'`, une autre dans `$&` et le reste dans `$'`. Le ralentissement est d'autant plus perceptible que les chaînes sont longues et que les motifs sont reconnus. Donc, évitez `$&`, `$'` et `$'` si vous le pouvez. Et si vous ne le pouvez pas, une fois que vous les avez utilisées, utilisez-les tant que vous voulez, car vous en avez déjà payé le prix. Souvenez-vous que certains algorithmes les apprécient vraiment. À partir de la version 5.005, la variable `$&` n'a plus le même coût que les deux autres.

Depuis Perl 5.6.1, les variables spéciales `@-` et `@+` peuvent remplacer fonctionnellement `$'`, `$&` et `$'`. Ces tableaux contiennent des pointeurs vers le début et la fin de chaque partie reconnue (voir *perlvar* pour tous les détails). Ils vous donnent donc bien accès aux mêmes informations mais sans l'inconvénient des copies de chaînes inutiles.

### 23.1.19 À quoi peut bien servir `\G` dans une expression rationnelle ?

On peut utiliser l'ancre `\G` pour débiter une reconnaissance dans une chaîne là où la recherche précédente s'est arrêtée. Le moteur d'expression rationnelle ne peut pas sauter de caractère pour trouver la prochaine reconnaissance avec cette ancre. Donc `<\G>` est similaire à l'ancre de début de chaîne, `^`. L'ancre `\G` est habituellement utilisée en conjonction avec le modificateur `/g`. Elle utilise la valeur de `pos()` comme position de départ de la prochaine reconnaissance. Au fur et à mesure que l'opérateur de reconnaissance fait des reconnaissances, il met à jour `pos()` avec la position du caractère suivant la dernière reconnaissance (ou le premier caractère de la prochaine reconnaissance selon le point de vue qu'on adopte). Chaque chaîne a sa propre valeur `pos()`.

Supposons que vous vouliez reconnaître toutes les paires consécutives de chiffres dans une chaîne telle que "1122a44" et vous arrêtez dès que vous rencontrez autre chose qu'un chiffre. Vous voulez donc reconnaître 11 et 22 mais la lettre a apparaissant entre 22 et 44 doit vous stopper là. La simple reconnaissance de paires de chiffres passerait au-delà du a et reconnaîtrait 44.

```
$_ = "1122a44";
my @paires = m/(\d\d)/g; # qw(11 22 44)
```

Si vous utilisez l'ancre `\G`, vous forcez la reconnaissance suivant 22 à débiter avec `a`. L'expression rationnelle ne peut donc être reconnue puisqu'elle ne trouve pas un chiffre et donc la prochaine reconnaissance échoue et l'opérateur de reconnaissance retourne la liste des paires déjà trouvées.

```
$_ = "1122a44";
my @paires = m/\G(\d\d)/g; # qw(11 22)
```

Vous pouvez aussi utiliser l'ancre `\G` dans un contexte scalaire. Il faut alors utiliser le modificateur `/g`.

```
$_ = "1122a44";
while(m/\G(\d\d)/g)
{
 print "Vu $1\n";
}
```

Après un échec de la reconnaissance sur la lettre `a`, perl réinitialise `pos()` et la prochaine recherche dans la même chaîne débutera à nouveau au début.

```
$_ = "1122a44";
while(m/\G(\d\d)/g)
{
 print "Vu $1\n";
}

print "Vu $1 après while" if m/(\d\d)/g; # trouve "11"
```

Vous pouvez désactiver la réinitialisation de `pos()` lors de l'échec d'une reconnaissance en utilisant le modificateur `/c`. La prochaines reconnaissances débuteront toutes là où s'est terminée la dernière reconnaissance (la valeur de `pos()`) même si une reconnaissance sur la même chaîne à échouer entre temps. Dans notre cas, la reconnaissance après la boucle `while()` commencera au `a` (là où s'est terminée la dernière reconnaissance) et puisqu'elle n'utiliser aucune ancre, elle pourra sauter le `a` et trouver "44".

```
$_ = "1122a44";
while(m/\G(\d\d)/gc)
{
 print "Vu $1\n";
}

print "Vu $1 après while" if m/(\d\d)/g; # trouve "44"
```

Typiquement, on utilise l'ancre `\G` avec le modificateur `/c` lorsqu'on souhaite pouvoir essayer un autre motif si celui-ci échoue, comme dans analyseur syntaxique. Jeffrey Friedl nous offre cet exemple qui fonctionne à partir de la version 5.004.

```
while (<>) {
 chomp;
 PARSER: {
 m/ \G(\d+\b)/gcx && do { print "nombre: $1\n"; redo; };
 m/ \G(\w+)/gcx && do { print "mot: $1\n"; redo; };
 m/ \G(\s+)/gcx && do { print "espace: $1\n"; redo; };
 m/ \G([^\w\d]+)/gcx && do { print "autre: $1\n"; redo; };
 }
}
```

Pour chaque ligne, la boucle `PARSER` essaie de reconnaître une série de chiffres suivies d'une limite de mot. Cette reconnaissance doit débiter là où s'est arrêtée la précédente reconnaissance (où au début de la chaîne la première fois). Puisque `m/ \G( \d+\b )/gcx` utilise le modificateur `</c>`, même si la chaîne n'est pas reconnue par l'expression rationnelle, perl ne réinitialise pas `pos()` et essaie le prochain motif en démarrant à la même position.

### 23.1.20 Les expressions rationnelles de Perl sont-elles AFD ou AFN ? Sont-elles conformes à POSIX ?

Bien qu'il soit vrai que les expressions rationnelles de Perl ressemblent aux AFD (automate fini déterminé) du programme `egrep(1)`, elles sont dans les faits implémentées comme AFN (automate fini indéterminé) pour permettre le retour en arrière et la mémorisation de sous-motif. Et elles ne sont pas non plus conformes à POSIX, car celui-ci garantit le comportement du pire dans tous les cas. (Il semble que certains préfèrent une garantie de cohérence, même quand ce qui est garanti est la lenteur). Lisez le livre "Mastering Regular Expressions" (from O'Reilly) par Jeffrey Friedl pour tous les détails que vous pouvez espérer connaître sur ces matières (la référence complète est dans *perlfaq2*).

### 23.1.21 Qu'est ce qui ne va pas avec l'utilisation de `grep` dans un contexte vide ?

Le problème c'est que `grep` construit une liste comme résultat, quel que soit le contexte. Cela signifie que vous amenez Perl à construire une liste que vous allez jeter juste après. Si cette liste est grosse, vous gêchez du temps et de l'espace mémoire. Si votre objectif est de parcourir la liste alors utiliser une boucle `foreach` pour cela.

Dans les versions de Perl antérieures à la version 5.8.1, `map` souffrait du même problème. Mais depuis la version 5.8.1, cela a été corrigé et `map` tient compte maintenant de son contexte - dans un contexte vide, aucune liste n'est construite.

### 23.1.22 Comment reconnaître des chaînes avec des caractères multi-octets ?

Depuis la version 5.6, Perl gère les caractères multi-octets avec une qualité qui s'accroît de version en version. Perl 5.8 ou plus est recommandé. Le support des caractères multi-octets inclut Unicode et les anciens encodages à travers le module `Encode`. Voir *perluniintro*, *perlunicode* et *Encode*.

Si vous utilisez une vieille version de Perl, vous pouvez gérer l'Unicode via le module `Unicode::String` et les conversion de caractères en utilisant les modules `Unicode::Map8` and `Unicode::Map`. Si vous utilisez les encodages japonais, vous pouvez essayer `jperl 5.005_03`.

Pour finir, Jeffrey Friedl qui a consacré à ce sujet un article dans le numéro 5 de *The Perl Journal*, vous offre les approches suivantes.

Supposons que vous ayez un codage de ces mystérieux Martiens où les paires de lettre majuscules ASCII codent une lettre simple martienne (i.e. les 2 octets "CV" donne une simple lettre martienne, ainsi que "SG", "VS", "XX", etc.). D'autres octets représentent de simples caractères comme l'ASCII.

Ainsi, la chaîne martienne "Je suis CVSGXX!" utilise 15 octets pour coder les 12 caractères 'J', 'e', ' ', 's', 'u', 'i', 's', ' ', 'CV', 'SG', 'XX', '!'.  
' , 'CV', 'SG', 'XX', '!'.

Maintenant, vous voulez chercher le simple caractère `/GX/`. Perl ne connaît rien au martien, donc il trouvera les 2 octets "GX" dans la chaîne "Je suis CVSGXX!", alors que ce caractère n'y est pas: il semble y être car "SG" est à côté de "XX", mais il n'y a pas de réel "GX". C'est un grand problème.

Voici quelques manières, toutes pénibles, de traiter cela :

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # garantit que les octets "martiens"
 # ne sont plus contigus
print "GX trouvé!\n" if $martian =~ /GX/;
```

Ou ainsi :

```
@chars = $martian =~ m/([A-Z][A-Z]|^[^A-Z])/g;
c'est conceptuellement similaire à: @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
 print "GX trouvé!\n", last if $char eq 'GX';
}
```

Ou encore ainsi :

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) { # \G probablement inutile
 print "GX trouvé!\n", last if $1 eq 'GX';
}
```

Voici une autre solution proposée par Benjamin Goldberg et qui utilise une assertion négative de longueur nulle pour regarder en arrière.

```
print "found GX!\n" if $martian =~ m/
 (?<![A-Z])
 (?:[A-Z][A-Z])*?
 GX
/x;
```

L'expression rationnelle est reconnue si le caractère "martien" GX est présent dans la chaîne et échoue sinon.

Si vous ne souhaitez pas utiliser (?<!), une assertion négative arrière, vous pouvez remplacer (?<![A-Z]) par (?:^[^A-Z]). Cela a pour défaut de placer de mauvaises valeurs dans \$-[0] et \$+[0] mais, habituellement, on peut faire avec.

### 23.1.23 Comment rechercher un motif fourni par l'utilisateur ?

Eh bien, si c'est vraiment un motif, alors utilisez juste

```
chomp($pattern = <STDIN>);
if ($line =~ /$pattern/) { }
```

Ou, puisque vous n'avez aucune garantie que votre utilisateur a entré une expression rationnelle valide, piègez une éventuelle exception de cette façon :

```
if (eval { $line =~ /$pattern/ }) { }
```

Si vous voulez seulement chercher une chaîne et non pas un motif, alors vous devriez soit utiliser la fonction `index()`, qui est faite pour cela, soit, s'il est impossible de vous convaincre de ne pas utiliser une expression rationnelle pour autre chose qu'un motif, assurez-vous au moins d'utiliser `\Q...\E`, documenté dans *perlre*.

```
$pattern = <STDIN>;
open (FILE, $input) or die "Couldn't open input $input: $!; aborting";
while (<FILE>) {
 print if /\Q$pattern\E/;
}
close FILE;
```

## 23.2 AUTEUR ET COPYRIGHT

Copyright (c) 1997-1999 Tom Christiansen, Nathan Torkington et autres auteurs cités ci-dessus. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même. Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code précisant l'origine serait de bonne courtoisie mais n'est pas indispensable.

## 23.3 TRADUCTION

La traduction française est distribuée avec les mêmes droits que sa version originale (voir ci-dessus).

### 23.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 23.3.2 Traducteur

Marianne Roger <maroger@maretmanu.org>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit (paul.gaborit at enstimac.fr).

### 23.3.3 Relecture

Régis Julié (Regis.Julie@cetelem.fr), Gérard Delafond.

# Chapitre 24

## perlfaq7

Questions générales sur le langage Perl

### 24.1 DESCRIPTION

Cette section traite des questions générales sur le langage Perl qui ne trouvent leur place dans aucune autre section.

#### 24.1.1 Puis-je avoir une BNF/yacc/RE pour le langage Perl ?

Il n'y a pas de BNF, mais vous pouvez vous frayer un chemin dans la grammaire de yacc dans `perly.y` au sein de la distribution source si vous êtes particulièrement brave. La grammaire repose sur un code de mots-clés complexe, préparez-vous donc à vous aventurer aussi dans `toke.c`.

Selon les termes de Chaim Frenkel : "La grammaire de Perl ne peut pas être réduite à une BNF. Le travail d'analyse de perl est distribué entre yacc, l'analyser lexical, de la fumée et des miroirs".

#### 24.1.2 Quels sont tous ces \$ @%&\* de signes de ponctuation, et comment savoir quand les utiliser ?

Ce sont des spécificateurs, comme détaillé dans *perldata* :

- \$ pour les valeurs scalaires (nombres, chaînes ou références)
- @ pour les tableaux
- % pour les hachages (tableaux associatifs)
- & pour les sous-programmes (alias les fonctions, les procédures, les méthodes)
- \* pour tous les types de la table des noms courante. Dans la version 4, vous les utilisiez comme des pointeurs, mais dans les versions modernes de perl, vous pouvez utiliser simplement les références.

Les trois autres symboles que vous risquez fortement de rencontrer sans qu'ils soient véritablement des spécificateurs de type sont :

- <> utilisés pour récupérer un enregistrement depuis un filehandle.
- \ prend une référence à quelque chose.

Notez que <FICHIER> n'est *ni* le spécificateur de type pour les fichiers *ni* le nom du handle. C'est l'opérateur <> appliqué au handle FICHIER. Il lit une ligne (en fait un enregistrement – voir `$/` in *perlvar*) depuis le handle FICHIER dans un contexte scalaire, ou *toutes* les lignes dans un contexte de liste. Lorsqu'on ouvre, ferme ou réalise n'importe quelle autre opération à part <> sur des fichiers, ou même si l'on parle du handle, n'utilisez *pas* les crochets. Ces expressions sont correctes : `eof(FH)`, `seek(FH, 0, 2)` et "copie de STDIN vers FICHIER".

### 24.1.3 Dois-je toujours/jamais mettre mes chaînes entre guillemets ou utiliser les points-virgules et les virgules ?

Normalement, un bareword n'a pas besoin d'être mis entre guillemets, mais dans la plupart des cas, il le devrait probablement (et doit l'être sous `use strict`). Mais une clé de hachage constituée d'un simple mot (qui ne soit pas le nom d'un sous-programme) et l'opérande de gauche de l'opérateur `=>`, comptent tous les deux comme s'ils étaient entre guillemets :

|                              |                                |
|------------------------------|--------------------------------|
| Ceci                         | équivalent à ceci              |
| -----                        | -----                          |
| <code>\$foo{line}</code>     | <code>\$foo{'line'}</code>     |
| <code>bar =&gt; stuff</code> | <code>'bar' =&gt; stuff</code> |

Le point-virgule final dans un bloc est optionnel, tout comme la virgule finale dans une liste. Un bon style (voir *perlstyle*) préconise de les mettre sauf dans les expressions d'une seule ligne :

```
if ($whoops) { exit 1 }
@nums = (1, 2, 3);

if ($whoops) {
 exit 1;
}
@lines = (
 "There Beren came from mountains cold",
 "And lost he wandered under leaves",
);
```

### 24.1.4 Comment ignorer certaines valeurs de retour ?

Une façon est de traiter les valeurs de retour comme une liste et de les indexer :

```
$dir = (getpwnam($user))[7];
```

Une autre façon est d'utiliser `undef` comme un élément de la partie gauche :

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Vous pouvez aussi utiliser une tranche de liste pour sélectionner uniquement les éléments nécessaires :

```
($dev, $ino, $uid, $gid) = (stat($file))[0,1,4,5];
```

### 24.1.5 Comment bloquer temporairement les avertissements ?

Si vous utilisez Perl 5.6.0 ou supérieur, le pragma `use warnings` vous permet un contrôle fin des avertissements qui sont générés. Voir *perllexwarn* pour plus de détails.

```
{
 no warnings; # désactive temporairement les avertissements
 $a = $b + $c; # je sais qu'elles peuvent être indéfinies
}
```

De plus, vous pouvez activer ou désactiver les avertissements par catégories. Vous désactivez les catégories que vous souhaitez ignorer tout en conservant les autres catégories d'avertissements. Voir *perllexwarn* pour tous les détails dont les noms des catégories et leur hiérarchie.

```
{
 no warnings 'uninitialized';
 $a = $b + $c;
}
```

Si vous avez une version plus ancienne de Perl, la variable `$_W` (documentée dans *perlvar*) contrôle les avertissements lors de l'exécution d'un bloc :

```
{
 local $_W = 0; # supprimer temporairement les
 # avertissements
 $a = $b + $c; # je sais que ces variables sont
 # peut-être indéfinies
}
```

Notez que comme toutes les variables de ponctuation, vous ne pouvez pas utiliser `my()` sur `$_W`, uniquement `local()`.

### 24.1.6 Qu'est-ce qu'une extension ?

Une extensions est une façon d'appeler depuis Perl du code C compilé. Lire *perlxsut* est une bonne façon d'en apprendre plus sur les extensions.

### 24.1.7 Pourquoi les opérateurs de Perl ont-ils une précedence différente de celle des opérateurs en C ?

En fait, ce n'est pas le cas. Tous les opérateurs C que Perl copie ont la même précedence en Perl qu'en C. Le problème est avec les opérateurs que C ne possède pas, en particulier les fonctions qui donnent un contexte de liste à tout ce qui se trouve à leur droite, par exemple `print`, `chmod`, `exec` et ainsi de suite. De telles fonctions sont appelées des "opérateurs de liste" et apparaissent en tant que tels dans la table de précedence de *perlop*.

Une erreur commune est d'écrire :

```
unlink $file || die "snafu";
```

Qui est interprété ainsi :

```
unlink ($file || die "snafu");
```

Pour éviter ce problème, placez soit des parenthèses supplémentaires ou utilisez l'opérateur de précedence super basse `or` :

```
(unlink $file) || die "snafu";
unlink $file or die "snafu";
```

Les opérateurs "anglais" (`and`, `or`, `xor`, et `not`) ont une précedence délibérément plus basse que celle des opérateurs de liste juste pour des situations telles que ci-dessus.

Un autre opérateur ayant une précedence surprenante est l'exponentiation. Il se lie encore plus étroitement que même le moins unaire, faisant du produit  $-2^{*2}$  un quatre négatif et non pas positif. Il s'associe aussi à droite, ce qui signifie que  $2^{*3}^{*2}$  vaut deux à la puissance neuvième, et non pas huit au carré.

Bien qu'il ait la même précedence qu'en C, l'opérateur `?:` de Perl produit une lvalue. Ce qui suit affecte `$x` soit à `$a` soit à `$b`, selon la valeur de vérité de `$maybe` :

```
($maybe ? $a : $b) = $x;
```

### 24.1.8 Comment déclarer/créer une structure ?

En général, vous ne "déclarez" pas une structure. Utilisez juste une référence de hachage (probablement anonyme). Voir *perlref* et *perldsc* pour plus de détails. Voici un exemple :

```
$person = {}; # nouveau hachage anonyme
$person->{AGE} = 24; # fixe le champ AGE à 24
$person->{NAME} = "Nat"; # fixe le champ NAME à "Nat"
```

Si vous recherchez quelque chose d'un peu plus rigoureux, essayez *perltoot*.



### 24.1.9 Comment créer un module ?

(contribution de brian d foy)

*perlmod*, *perlmodlib* et *perlmodstyle* expliquent les modules dans tous leurs détails. *perlnewmod* fournit un bref aperçu de la manière de procéder avec quelques suggestions de style.

Si vous avez besoin d'inclure du code C ou une interface vers une bibliothèque C dans votre module, vous aurez besoin de *h2xs*. *h2xs* créera pour vous la structure de distribution de votre module et les fichiers d'interface initiaux dont vous aurez besoin. *perlx*s et *perlxstut* donnent tous les détails.

Si vous n'avez pas besoin de code C, d'autres outils comme `ExtUtils::ModuleMaker` et `Module::Starter` peuvent vous aider à créer le squelette de distribution de votre module.

Vous pouvez aussi lire "Writing Perl Modules! for CPAN" (<http://apress.com/book/bookDisplay.html?bID=14>) par Sam Tregar qui est le meilleur guide pour créer une distribution de module.

### 24.1.10 Comment créer une classe ?

Voir *perloot* pour une introduction aux classes et aux objets, ainsi que *perlobj* et *perlbot*.

### 24.1.11 Comment déterminer si une variable est souillée ?

Vous pouvez utiliser la fonction `tainted()` du module `Scalar::Util` disponible sur CPAN (et inclus dans la distribution Perl depuis la version 5.8.0). Voir aussi Blanchiment et détection des données souillées in *perlsec*.

### 24.1.12 Qu'est-ce qu'une fermeture ?

Les fermetures sont documentées dans *perlref*.

La *Fermeture* est un terme d'informatique ayant un sens précis mais difficile à expliquer. Les fermetures sont implémentées en Perl par des sous-programmes anonymes ayant des références à des variables lexicales qui persistent hors de leur propre portées. Ces lexicaux se réfèrent de façon magique aux variables qui se trouvaient dans le coin lorsque le sous-programme a été défini (deep binding).

Les fermetures ont un sens dans tous les langages de programmation où la valeur de retour d'une fonction peut être une fonction, comme c'est le cas en Perl. Notez que certains langages fournissent des fonctions anonymes sans être capables de fournir des fermetures proprement dites ; le langage Python, par exemple. Pour plus d'informations sur les fermetures, regardez dans un livre sur la programmation fonctionnelle. Scheme est un langage qui non seulement supporte mais aussi encourage les fermetures.

Voici une fonction classique de génération de fonctions :

```
sub add_function_generator {
 return sub { shift() + shift() };
}

$add_sub = add_function_generator();
$sum = $add_sub->(4,5); # $sum vaut maintenant 9.
```

La fermeture fonctionne comme un *modèle de fonction* ayant des possibilités de personnalisation qui seront définies plus tard. Le sous-programme anonyme retourné par `add_function_generator()` n'est pas techniquement une fermeture car il ne se réfère à aucun lexical hors de sa propre portée.

Comparez cela à la fonction suivante, `make_adder()`, dans laquelle la fonction anonyme retournée contient une référence à une variable lexicale se trouvant hors de la portée de la fonction elle-même. Une telle référence nécessite que Perl renvoie une fermeture proprement dite, verrouillant ainsi pour toujours la valeur que le lexical avait lorsque la fonction fut créé.

```
sub make_adder {
 my $addpiece = shift;
 return sub { shift() + $addpiece };
}
```

```
$f1 = make_adder(20);
$f2 = make_adder(555);
```

Maintenant `&$f1($n)` vaut toujours 20 plus la valeur de `$n` que vous lui passez, tandis que `&$f2($n)` vaut toujours 555 plus ce qui se trouve dans `$n`. La variable `$addpiece` dans la fermeture persiste.

Les fermetures sont souvent utilisées pour des motifs moins ésotériques. Par exemple, lorsque vous désirez passer un morceau de code à une fonction :

```
my $line;
timeout(30, sub { $line = <STDIN> });
```

Si le code à exécuter a été passé en tant que chaîne, `'$line = <STDIN>'`, la fonction hypothétique `timeout()` n'aurait aucun moyen d'accéder à la variable lexicale `$line` dans la portée de son appelant.

### 24.1.13 Qu'est-ce que le suicide de variable et comment le prévenir ?

Ce problème a été corrigé dans perl 5.004\_05. Donc la prévention passe par la mise à jour de perl. ;)

Le suicide de variable se produit lorsque vous perdez la valeur d'une variable (temporairement ou de façon permanente). Il est causé par les interactions entre la portée définie par `my()` et `local()` et soit des fermetures soit des variables d'itération `foreach()` aliasées et des arguments de sous-programme (Note au lecteur : un peu bordélique, cette phrase). Il a été facile de perdre par inadvertance la valeur d'une variable de cette façon, mais désormais c'est bien plus difficile. Considérez ce code :

```
my $f = 'foo';
sub T {
 while ($i++ < 3) { my $f = $f; $f .= $i; print $f, "\n" }
}
T;
print "Finally $f\n";
```

Si vous constatez un suicide de variable, c'est que `my $f` dans le sous-programme ne récupère pas une nouvelle copie de `$f` dont la valeur est `foo`. La sortie suivante montre qu'à l'intérieur du sous-programme la valeur de `$f` est perdue alors qu'elle ne devrait pas :

```
foobar
foobarbar
foobarbarbar
Finally foo
```

Le `$f` qui se voit ajouter "bar" par trois fois devrait être un nouveau `$f`. `my $f` devrait créer une nouvelle variable locale chaque fois que la boucle est parcourue. La sortie normale est :

```
foobar
foobar
foobar
Finally foo
```

### 24.1.14 Comment passer/renvoyer {une fonction, un handle de fichier, un tableau, un hachage, une méthode, une expression rationnelle} ?

À l'exception des expressions rationnelles, vous devez passer des références à ces objets. Voir [Passage par Référence](#) in *perlsyntax* pour cette question particulière, et *perlref* pour plus d'informations sur les références.

Voir "Passer des expressions rationnelles", plus bas, pour le passage des expressions rationnelles.

#### Passer des variables et des fonctions

Les variables normales et les fonctions sont faciles à passer : passez juste une référence à une variable ou une fonction, existante ou anonyme :

```
func(\some_scalar);
```

```

func(\@some_array);
func([1 .. 10]);

func(\%some_hash);
func({ this => 10, that => 20 });

func(\&some_func);
func(sub { $_[0] ** $_[1] });

```

### Passer des handles de fichiers

Depuis Perl 5.6, vous pouvez représenter les handles de fichiers par de simples variables scalaires qui se traitent donc comme tous les autres scalaires.

```

open my $fh, $filename or die "Cannot open $filename! $!";
func($fh);

sub func {
 my $passed_fh = shift;

 my $line = <$passed_fh>;
}

```

Avant Perl 5.6, vous devez utiliser les notations `*FH` ou `\*FH`. Ce sont des "typeglobs" - voir [Typeglobs](#) et [handles de fichiers](#) in *perldata* et en particulier [Passage par référence](#) in *perlsub* pour plus d'information.

### Passer des expressions rationnelles

Pour passer des expressions rationnelles, vous devrez utiliser une version de Perl suffisamment récente pour supporter la construction `qr//`, passer des chaînes et utiliser un `eval` capturant les exceptions, soit encore vous montrer très très intelligent.

Voici un exemple montrant comment passer une chaîne devant être comparée comme expression rationnelle :

```

sub compare($$) {
 my ($val1, $regex) = @_;
 my $retval = $val1 =~ /$regex/;
 return $retval;
}

$match = compare("old McDonald", qr/d.*D/i);

```

Notez comment `qr//` autorise les drapeaux terminaux. Ce motif a été compilé à la compilation, même s'il n'a été exécuté que plus tard. La sympathique notation `qr//` n'a pas été introduite avant la version 5.005. Auparavant, vous deviez approcher ce problème bien moins intuitivement. La revoici par exemple si vous n'avez pas `qr//` :

```

sub compare($$) {
 my ($val1, $regex) = @_;
 my $retval = eval { $val1 =~ /$regex/ };
 die if $@;
 return $retval;
}

$match = compare("old McDonald", q/($?i)d.*D/);

```

Assurez-vous de ne jamais dire quelque chose comme ceci :

```
return eval "\$val =~ /$regex/"; # FAUX
```

ou quelqu'un pourrait glisser des séquences d'échappement du shell dans l'expressions rationnelle à cause de la double interpolation de l'`eval` et de la chaîne doublement mise entre guillemets. Par exemple :

```

$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';
eval "\$string =~ /$pattern_of_evil/";

```

Ceux qui préfèrent être très très intelligents peuvent voir le livre O'Reilly *Mastering Regular Expressions*, par Jeffrey Friedl. La fonction `Build_MatchMany_Function()` à la page 273 est particulièrement intéressante. Une citation complète de ce livre est donnée dans [perlfaq2](#).

### Passer des Méthodes

Pour passer une méthode objet à un sous-programme, vous pouvez faire ceci :

```

call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
 my ($count, $widget, $trick) = @_;
 for (my $i = 0; $i < $count; $i++) {
 $widget->$trick();
 }
}

```

Ou bien vous pouvez utiliser une fermeture pour encapsuler l'objet, son appel de méthode et ses arguments :

```

my $whatnot = sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
 my $code = shift;
 &$code();
}

```

Vous pourriez aussi étudier la méthode `can()` de la classe `UNIVERSAL` (qui fait partie de la distribution standard de perl).

### 24.1.15 Comment créer une variable statique ?

(contribution de brian d foy)

Perl n'a pas de variable "statique" qui ne serait accessible que dans la fonction où elle serait déclarée. Mais vous pouvez obtenir le même effet en utilisant des variables lexicales.

Vous pouvez simuler une variable statique en utilisant une variable lexicale qui se met hors de portée. Dans l'exemple ci-dessous, nous définissons un sous-programme `compteur` qui utilise la variable lexicale `$compte`. Puisque tout cela se fait dans un bloc `BEGIN`, non seulement `$compte` est défini lors de la compilation mais aussi elle se met hors de portée à la fin du bloc `BEGIN`. Le bloc `BEGIN` garantit que le sous-programme et la valeur qu'il utilise sont définis lors de la compilation et donc le sous-programme est utilisable comme n'importe quel autre sous-programme, et vous pouvez placer ce bout de code aux mêmes endroits qu'un sous-programme classique (en fin de programme par exemple). Le sous-programme `compteur` conserve une référence à la donnée et c'est le seul moyen que vous avez d'accéder à cette valeur (et à chaque fois que vous le faites, vous incrémentez la valeur). La donnée définie par `$compte` est donc une donnée privée de `compteur`.

```

BEGIN {
 my $compte = 1;
 sub compteur { $compte++ }
}

my $debut = compteur();

.... # du code qui appelle compteur();

my $fin = compteur();

```

Dans l'exemple précédent, nous avons créé une variable privée visible d'une seule fonction. Nous pourrions définir plusieurs fonctions tant que la variable reste visible et toutes ces fonctions partageraient la même variable "privée". Ce n'est plus réellement "statique" puisque plusieurs fonctions accèdent à la même variable. Dans l'exemple suivant, `incrimente_compteur` et `donne_compteur` partagent la variable. La première fonction ajoute 1 à la valeur alors que la seconde ne fait que retourner la valeur. Toutes deux peuvent accéder à la valeur et il n'existe pas d'autres moyens de le faire.

```

BEGIN {
 my $compte = 1;
 sub incremente_compteur { $compte++ }
 sub donne_compteur { $compte }
}

```

Pour déclarer une variable privée d'un fichier, vous pouvez encore utiliser une variable lexicale. La portée d'une variable lexicale se limite à son contenant (un bloc ou un fichier) et donc une variable lexicale déclarée dans un fichier est invisible depuis un autre fichier.

Voir *Variables privées persistantes* in *perlsub* pour plus de détails. Tout ce qui est présenté dans *perlref* concernant les fermetures peut aussi vous aider même si nous n'utilisons pas de sous-programmes anonymes dans cette réponse.

### 24.1.16 Quelle est la différence entre la portée dynamique et lexicale (statique) ? Entre local() et my() ?

local(\$x) sauvegarde l'ancienne valeur de la variable globale \$x et lui affecte une nouvelle valeur pour la durée du sous-programme, *valeur qui est visible dans les autres fonctions appelées depuis ce sous-programme*. C'est fait lors de l'exécution et donc cela s'appelle une portée dynamique. local() affecte toujours les variables globales, aussi appelées variables de paquetage ou variables dynamiques.

my(\$x) crée une nouvelle variable qui n'est visible que dans le sous-programme. C'est fait lors de la compilation et donc cela s'appelle une portée lexicale ou statique. my() affecte toujours les variables privées, aussi appelées variables lexicales ou (de façon impropre) variables (de portée) statique.

Par exemple :

```
sub visible {
 print "var has value $var\n";
}

sub dynamic {
 local $var = 'local'; # nouvelle valeur temporaire pour la
 visible(); # variable toujours globale appelée $var
}

sub lexical {
 my $var = 'private'; # nouvelle variable privée, $var
 visible(); # (invisible hors de la portée de la routine)
}

$var = 'global';

visible(); # affiche global
dynamic(); # affiche local
lexical(); # affiche global
```

Notez que, comme à aucun moment la valeur "private" n'est affichée. C'est parce que \$var n'a cette valeur qu'à l'intérieur du bloc de la fonction lexical(), et est cachée des sous-programmes appelés.

En résumé, local() ne crée pas comme vous pourriez le penser des variables locales, privées. Il donne seulement à une variable globale une valeur temporaire. my() est ce que vous recherchez si vous voulez des variables privées.

Voir Variables privées via my() in *perlsub* et Valeurs temporaires via local() in *perlsub* pour plus de détails.

### 24.1.17 Comment puis-je accéder à une variable dynamique lorsqu'un lexical de même nom est à portée ?

Si vous connaissez le nom du paquetage, vous pouvez tout simplement le mentionner explicitement comme dans \$Un\_paquetage::var. Notez que la notation \$::var ne signifie **pas** la variable dynamique \$var du paquetage courant mais celle du paquetage "main" exactement comme si vous aviez écrit \$main::var.

```
use vars '$var';
local $var = "global";
my $var = "lexical";

print "lexical is $var\n";
print "global is $main::var\n";
```

Vous pouvez aussi utiliser la directive our() du compilateur pour amener un vars dynamique à portée.

```
require 5.006; # our() n'existait avant 5.6
use vars '$var';

local $var = "global";
my $var = "lexical";

print "lexical is $var\n";

{
 our $var;
 print "global is $var\n";
}
```

### 24.1.18 Quelle est la différence entre les liaisons profondes et superficielles ?

Dans la liaison profonde, les variables lexicales mentionnées dans les sous-programmes anonymes sont les mêmes que celles qui étaient dans la portée lorsque le sous-programme fut créé. Dans la liaison superficielle, ce sont les variables, quelles qu'elles soient, ayant le même nom et s'étant trouvées dans la portée lorsque le sous-programme est appelé. Perl utilise toujours la liaison profonde des variables lexicales (i.e., celles créées avec `my()`). Toutefois, les variables dynamiques (aussi appelées globales, locales ou variables de paquetage) sont effectivement liées superficiellement. Considérez ceci comme simplement une raison de plus de ne pas les utiliser. Voir la réponse à Qu'est-ce qu'une fermeture ? (§24.1.12).

### 24.1.19 Pourquoi "my(\$ foo) = <FICHIER>;" ne marche pas ?

`my()` et `local()` donnent un contexte de liste à la partie droite de `=`. L'opération de lecture `<FH>`, comme tant d'autres fonctions et opérateurs de Perl, peut déterminer dans quel contexte elle a été appelée pour se comporter de façon appropriée. En général, la fonction `scalar()` peut aider. Cette fonction ne fait rien aux données elles-mêmes (contrairement au mythe populaire) mais dit seulement à ses arguments de se comporter à sa propre manière scalaire. Si cette fonction n'a pas de comportement scalaire défini, ceci ne vous aidera visiblement pas (tout comme avec `sort()`).

Pour forcer un contexte scalaire dans ce cas particulier, toutefois, vous devez tout simplement omettre les parenthèses :

```
local($foo) = <FILE>; # FAUX
local($foo) = scalar(<FILE>); # ok
local $foo = <FILE>; # bien
```

Vous devriez probablement utiliser des variables lexicales de toute façon, même si le résultat est le même ici :

```
my($foo) = <FILE>; # FAUX
my $foo = <FILE>; # bien
```

### 24.1.20 Comment redéfinir une fonction, un opérateur ou une méthode prédéfini ?

Pourquoi donc voulez-vous faire cela ?

Si vous voulez surcharger une fonction prédéfinie, telle qu'`open()`, alors vous devrez importer la nouvelle définition depuis un module différent. Voir *Surcharge des Fonctions Prédéfinies* in *perlsub*. Il y a aussi un exemple dans `Class::Template` in *perltoot*.

Si vous voulez surcharger un opérateur de Perl, tel que `+` ou `**`, alors vous voudrez utiliser le pragma `use overload`, documentée dans *overload*.

Si vous parlez d'obscurcir les appels de méthode dans les classes parent, voyez *Polymorphisme* in *perltoot*.

### 24.1.21 Quelle est la différence entre l'appel d'une fonction par `&foo` et par `foo()` ?

Quand vous appelez une fonction par `&foo`, vous permettez à cette fonction d'accéder à vos valeurs `@_` courantes, et vous court-circuitez les prototypes. Cela signifie que la fonction ne récupère pas un `@_` vide, mais le vôtre ! Même si l'on ne peut pas exactement parler d'un bug (c'est documenté de cette façon dans *perlsub*), il serait difficile de considérer cela comme une caractéristique dans la plupart des cas.

Quand vous appelez votre fonction par `&foo()`, alors vous obtenez *effectivement* une nouvelle `@_`, mais le prototypage est toujours court-circuité.

Normalement, vous préférerez appeler une fonction en utilisant `foo()`. Vous ne pouvez omettre les parenthèses que si la fonction est déjà connue par le compilateur parce qu'il a déjà vu sa définition (`use` mais pas `require`), ou via une référence en avant ou une déclaration `use subs`. Même dans ce cas, vous obtenez une `@_` propre sans aucune des vieilles valeurs suintant là où elles ne devraient pas.

### 24.1.22 Comment créer une instruction switch ou case ?

C'est expliqué plus en profondeur dans *perlsyn*. En bref, il n'existe pas d'instruction case officielle, à cause de la variété des tests possibles en Perl (comparaison numérique, comparaison de chaînes, comparaison de glob, appariement d'expression rationnelle, comparaisons surchargées...). Larry ne pouvait pas se décider sur la meilleure façon de faire cela, il a donc laissé tomber et c'est resté sur la liste des voeux pieux depuis perl1.

Depuis Perl 5.8, pour avoir les instruction switch et case, vous pouvez utiliser le module Switch en disant :

```
use Switch;
```

Ce n'est pas aussi rapide que cela pourrait l'être puisque ce n'est pas vraiment intégré au langage (c'est fait par du filtrage des sources) mais ça existe et c'est pratique.

Si vous préférez utiliser du Perl pur, la réponse générale est d'écrire une construction comme celle-ci :

```
for ($variable_to_test) {
 if (/pat1/) { } # faire quelque chose
 elsif (/pat2/) { } # faire quelque chose d'autre
 elsif (/pat3/) { } # faire quelque chose d'autre
 else { } # défaut
}
```

Voici un exemple simple de switch basé sur un appariement de motif, cette fois mis en page de façon à le faire ressembler un peu plus à une instruction switch. Nous allons créer un choix multiple basé sur le type de référence stockée dans \$whatchamacallit :

```
SWITCH: for (ref $whatchamacallit) {

 /^$/ && die "not a reference";

 /SCALAR/ && do {
 print_scalar($$ref);
 last SWITCH;
 };

 /ARRAY/ && do {
 print_array(@$ref);
 last SWITCH;
 };

 /HASH/ && do {
 print_hash(%$ref);
 last SWITCH;
 };

 /CODE/ && do {
 warn "can't print function ref";
 last SWITCH;
 };

 # DEFAULT

 warn "User defined type skipped";

}
```

Voir *perlsyn*/"BLOCs de base et instruction switch" pour d'autres exemples dans ce style.

Vous devrez parfois échanger les positions de la constante et de la variable. Par exemple, disons que vous voulez tester une réponse parmi de nombreuses vous ayant été données, mais d'une façon insensible à la casse qui permette aussi les abréviations. Vous pouvez utiliser la technique suivante si les chaînes commencent toutes par des caractères différents ou si vous désirez arranger les correspondances de façon que l'une ait la précedence sur une autre, tout comme "SEND" a la précedence sur "STOP" ici :

```

chomp($answer = <>);
if ("SEND" =~ /^Q$answer/i) { print "Action is send\n" }
elsif ("STOP" =~ /^Q$answer/i) { print "Action is stop\n" }
elsif ("ABORT" =~ /^Q$answer/i) { print "Action is abort\n" }
elsif ("LIST" =~ /^Q$answer/i) { print "Action is list\n" }
elsif ("EDIT" =~ /^Q$answer/i) { print "Action is edit\n" }

```

Une approche totalement différente est de créer une référence de hachage ou de fonction.

```

my %commands = (
 "happy" => \&joy,
 "sad", => \&sullen,
 "done" => sub { die "See ya!" },
 "mad" => \&angry,
);

print "How are you? ";
chomp($string = <STDIN>);
if ($commands{$string}) {
 $commands{$string}->();
} else {
 print "No such command: $string\n";
}

```

### 24.1.23 Comment intercepter les accès aux variables, aux fonctions, aux méthodes indéfinies ?

La méthode AUTOLOAD, dont il est question dans Autochargement in *perlsub* et AUTOLOAD : les méthodes mandataires in *perltoot*, vous permet de capturer les appels aux fonctions et aux méthodes indéfinies.

Quant aux variables indéfinies qui provoqueraient un avertissement si `use warnings` est actif, il vous suffit de transformer l'avertissement en erreur.

```
use warnings FATAL => qw(uninitialized);
```

### 24.1.24 Pourquoi une méthode incluse dans ce même fichier ne peut-elle pas être trouvée ?

Quelques raisons possibles : votre héritage se trouble, vous avez fait une faute de frappe dans le nom de la méthode, ou l'objet est d'un mauvais type. Regardez dans *perltoot* pour des détails là-dessus. Vous pourriez aussi utiliser `print ref($object)` pour déterminer par quelle classe `$object` a été consacré.

Une autre raison possible de problèmes est que vous avez utilisé la syntaxe d'objet indirect (eg, `find Guru "Samy"`) sur le nom d'une classe avant que Perl n'ait vu qu'un tel paquetage existe. Il est plus sage de s'assurer que vos paquetages sont tous définis avant de commencer à les utiliser, ce qui sera fait si vous utilisez l'instruction `use` au lieu de `require`. Sinon, assurez-vous d'utiliser la notation fléchée à la place (eg, `Guru->find("Samy")`). La notation d'objet est expliquée dans *perlobj*.

Assurez-vous de vous renseigner sur la création des modules en lisant *perlmod* et les périls des objets indirects dans Appel de méthodes in *perlobj*.

### 24.1.25 Comment déterminer mon paquetage courant ?

Si vous êtes juste un programme pris au hasard, vous pouvez faire ceci pour découvrir quel est le paquetage compilé sur le moment :

```
my $packname = __PACKAGE__;
```

Mais si vous êtes une méthode et que vous voulez afficher un message d'erreur qui inclut le genre d'objet sur lequel vous avez été appelée (qui n'est pas nécessairement le même que celui dans lequel vous avez été compilée) :

```

sub amethod {
 my $self = shift;
 my $class = ref($self) || $self;
 warn "called me from a $class object";
}

```



### 24.1.26 Comment commenter un grand bloc de code perl ?

Vous pouvez utiliser POD pour le désactiver. Entourer le bloc à commenter de marqueurs POD. La directive `=begin` marque le début d'une section pour un formateur particulier. Utilisez le format `comment`, qu'aucun formateur ne risque de comprendre (par convention). Marquer la fin du bloc par `=end`.

```
le programme est ici

=begin comment

tout ce truc

ici sera ignoré
par tout le monde

=end commentaire

=cut

le programme continue
```

Les directives POD ne se placent pas n'importe où. Vous devez mettre une directive POD là où l'analyseur syntaxique attend une nouvelle instruction, pas juste au milieu d'une expression ou d'un autre élément grammaticale quelconque.

Voir *perlpod* pour plus de détails.

### 24.1.27 Comment supprimer un paquetage ?

Utilisez ce code, fourni par Mark-Jason Dominus:

```
sub scrub_package {
 no strict 'refs';
 my $pack = shift;
 die "Shouldn't delete main package"
 if $pack eq "" || $pack eq "main";
 my $stash = *{$pack . '::'}{HASH};
 my $name;
 foreach $name (keys %$stash) {
 my $fullname = $pack . '::' . $name;
 # Get rid of everything with that name.
 undef $$fullname;
 undef @$fullname;
 undef %$fullname;
 undef &$fullname;
 undef *$fullname;
 }
}
```

Ou, si vous utilisez une version récente de Perl, vous pouvez juste utiliser à la place la fonction `Symbol::delete_package()`.

### 24.1.28 Comment utiliser une variable comme nom de variable ?

Les débutants pensent souvent qu'ils veulent qu'une variable contienne le nom d'une variable.

```
$fred = 23;
$varname = "fred";
++$$varname; # $fred vaut maintenant 24
```

Ceci marche *parfois*, mais c'est une très mauvaise idée pour deux raisons.

La première raison est que cela *ne marche que pour les variables globales*. Cela signifie que si \$fred était une variable lexicale créée avec my() dans le code ci-dessus, alors le code ne marcherait pas du tout : vous accéderiez accidentellement à la variable globale et passeriez complètement par-dessus le lexical privé. Les variables globales sont mauvaises car elles peuvent facilement entrer en collision accidentellement et produisent en général un code non-extensible et confus.

Les références symboliques sont interdites sous le pragma use strict. Ce ne sont pas de vraies références et par conséquent elles ne sont pas comptées dans les références ni traitées par le ramasse-miettes.

La seconde raison qui rend mauvaise l'utilisation d'une variable pour contenir le nom d'une autre variable est que, souvent, le besoin provient d'un manque de compréhension des structures de données de Perl, en particulier des hachages. En utilisant des références symboliques, vous utilisez juste le hachage de la table de symboles du paquetage (comme %main: :) à la place d'un hachage défini par l'utilisateur. La solution est d'utiliser à la place votre propre hachage ou une vraie référence.

```
$USER_VARS{"fred"} = 23;
$varname = "fred";
$USER_VARS{$varname}++; # ce n'est pas $$varname++
```

Ici nous utilisons le hachage %USER\_VARS à la place de références symboliques. Parfois, cela est nécessaire quand on lit des chaînes auprès de l'utilisateur avec des références de variable et qu'on veut les étendre aux valeurs des variables du programme perl. C'est aussi une mauvaise idée car cela fait entrer en conflit l'espace de noms adressable par le programme et celui adressable par l'utilisateur. Au lieu de lire une chaîne et de la développer pour obtenir le contenu des variables de votre programme :

```
$str = 'this has a $fred and $barney in it';
$str =~ s/(\$\w+)/$1/eeg; # besoin d'un double eval
```

Il serait meilleur de conserver un hachage comme %USER\_VARS et d'avoir des références de variable pointant effectivement vers des entrées de ce hachage :

```
$str =~ s/\$(\w+)/$USER_VARS{$1}/g; # pas de /e du tout ici
```

C'est plus rapide, plus propre, et plus sûr que l'approche précédente. Bien sûr, vous n'avez pas besoin d'utiliser un dollar. Vous pourriez utiliser votre propre caractère pour rendre le tout moins confus, comme des symboles de pourcents, etc.

```
$str = 'this has a %fred% and %barney% in it';
$str =~ s/%(\w+)/$USER_VARS{$1}/g; # pas de /e du tout ici
```

Une autre raison qui amène les gens à croire qu'ils ont besoin qu'une variable contienne le nom d'une autre variable est qu'ils ne savent pas comment construire des structures de données correctes en utilisant des hachages. Par exemple, supposons qu'ils aient voulu deux hachages dans leur programme : %fred et %barney, et veuillent utiliser une autre variable scalaire pour s'y référer par leurs noms.

```
$name = "fred";
$name{WIFE} = "wilma"; # définit %fred

$name = "barney";
$name{WIFE} = "betty"; # définit %barney
```

C'est toujours une référence symbolique, avec toujours les problèmes énumérés ci-dessus sur les bras. Il serait bien meilleur d'écrire :

```
$folks{"fred"}{WIFE} = "wilma";
$folks{"barney"}{WIFE} = "betty";
```

Et de simplement utiliser un hachage multiniveau pour commencer.

La seule occasion où vous *devez* absolument utiliser des références symboliques est lorsque vous devez réellement vous référer à la table de symboles. Cela peut se produire parce qu'il s'agit de quelque chose dont on ne peut pas prendre une vraie référence, tel qu'un nom de format. Faire cela peut aussi être important pour les appels de méthodes, puisqu'ils passent toujours par la table de symboles pour leur résolution.

Dans ces cas, vous devrez supprimer temporairement les strict 'refs' de façon à pouvoir jouer avec la table de symboles. Par exemple :

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
 no strict 'refs'; # renege for the block
 *$name = sub { "@_" };
}
```

Toutes ces fonctions (red(), blue(), green(), etc.) apparaissent comme différentes, mais le vrai code dans la fermeture n'a effectivement été compilé qu'une fois.

Vous voudrez donc parfois utiliser les références symboliques pour manipuler directement la table de symboles. Cela n'a pas d'importance pour les formats, les handles, et les sous-programmes, car ils sont toujours globaux – vous ne pouvez pas utiliser my() sur eux. Mais pour les scalaires, les tableaux et les hachages – et habituellement pour les sous-programmes – vous préférerez probablement n'utiliser que de vraies références (les références dures).

### 24.1.29 Que signifie "bad interpreter" ?

(contribution de brian d foy)

Le message "bad interpreter" provient du shell et non de perl. Le message exact dépend de votre plateforme, du shell utilisé et de réglage des locale.

Si vous voyez "bad interpreter - no such file or directory", c'est que la première ligne de votre script (la ligne "shebang") ne contient pas un chemin correcte vers perl (ou n'importe quel programme capable de comprendre le script). Parfois cela arrive lorsque vous passez un script d'une machine à une autre dont le perl est installé différemment – /usr/bin/perl au lieu de /usr/local/bin/perl par exemple. Cela peut aussi indiquer que le fichier source contient CRLF comme fin de lignes alors que la machine s'attend uniquement à LF : le shell cherche /usr/bin/perl<CR> qu'il ne trouve évidemment pas.

Si vous voyez "bad interpreter: permission denied", il vous faut rendre votre script exécutable.

Dans tous les cas, vous devriez pouvoir faire exécuter explicitement le script par perl :

```
% perl script.pl
```

Si vous obtenez un message du genre "perl: command not found", alors perl n'est pas dans votre PATH ce qui signifie que perl n'est pas là où vous l'attendez et qu'il vous faudra corriger la ligne de "shebang".

## 24.2 AUTEUR ET COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen, Nathan Torkington et autres auteurs sus-cités. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même.

Contrairement à sa distribution, tous les exemples de code de ce fichier sont placés par le présent acte dans le domaine public. Vous avez l'autorisation et êtes encouragés à utiliser ce code dans vos propres programmes pour vous amuser ou pour gagner de l'argent selon votre humeur. Un simple commentaire dans le code créditant les auteurs serait courtois mais n'est pas requis.

## 24.3 TRADUCTION

### 24.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 24.3.2 Traducteur

Traduction initiale : Roland Trique <roland.trique@free.fr>. Mise à jour : Paul Gaborit <paul.gaborit@enstimac.fr>.

### 24.3.3 Relecture

Régis Julié <Regis.Julie@cetelem.fr>, Gérard Delafond.

# Chapitre 25

## perlfaq8

Interactions avec le système

### 25.1 DESCRIPTION

Cette section de la FAQ Perl traite des questions concernant les interactions avec le système d'exploitation. Cela inclut les mécanismes de communication inter-processus (IPC – Inter Process Communication en anglais), le pilotage de l'interface utilisateur (clavier, écran et souris), et d'une façon générale tout ce qui ne relève pas de la manipulation de données.

Lisez les FAQ et la documentation spécifique au portage de perl sur votre système d'exploitation (par ex. *perlvms*, *perlplan9*, etc.). Vous devriez y trouver de plus amples informations sur les spécificités de votre perl.

#### 25.1.1 Comment savoir sur quel système d'exploitation je tourne ?

La variable \$^O (\$OSNAME si vous utilisez le module English) contient une indication sur le nom du système d'exploitation (pas son numéro de version) sur lequel votre exécutable perl a été compilé.

#### 25.1.2 Pourquoi ne revient-on pas après un exec() ?

Parce que c'est ainsi qu'il fonctionne : il remplace le processus qui tourne par un nouveau. Si vous voulez continuer après (ce qui est probablement le cas si vous vous posez cette question), utilisez plutôt system().

#### 25.1.3 Comment utiliser le clavier/écran/souris de façon élaborée ?

La façon d'accéder aux (ou de contrôler les) claviers, écrans et souris ("mulots") dépend fortement du système d'exploitation. Essayez les modules suivants :

##### Clavier

|                      |                       |
|----------------------|-----------------------|
| Term::Cap            | Distribution standard |
| Term::ReadKey        | CPAN                  |
| Term::ReadLine::Gnu  | CPAN                  |
| Term::ReadLine::Perl | CPAN                  |
| Term::Screen         | CPAN                  |

##### Écran

|                 |                       |
|-----------------|-----------------------|
| Term::Cap       | Distribution standard |
| Curses          | CPAN                  |
| Term::ANSIColor | CPAN                  |

##### Souris

|    |      |
|----|------|
| Tk | CPAN |
|----|------|

Certains cas spécifiques sont examinés sous forme d'exemples dans d'autres réponses de cette FAQ.

### 25.1.4 Comment afficher quelque chose en couleur ?

En général, on ne le fait pas, parce qu'on ne sait pas si le receveur a un afficheur comprenant les couleurs. Cependant, si vous avez la certitude de trouver à l'autre bout un terminal ANSI qui traite la couleur, vous pouvez utiliser le module `Term::ANSIColor` de CPAN :

```
use Term::ANSIColor;
print color("red"), "Stop!\n", color("reset");
print color("green"), "Go!\n", color("reset");
```

Ou comme ceci :

```
use Term::ANSIColor qw(:constants);
print RED, "Stop!\n", RESET;
print GREEN, "Go!\n", RESET;
```

### 25.1.5 Comment lire simplement une touche sans attendre un appui sur "entrée" ?

Le contrôle des tampons en entrée est fortement dépendant du système. Sur la plupart d'entre eux, vous pouvez simplement utiliser la commande `stty` comme montré dans `getc` in *perlfunc*, mais visiblement, cela vous entraîne déjà dans les méandres de la portabilité.

```
open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY); # peut marcher
OU BIEN
sysread(TTY, $key, 1); # marche sans doute
system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```

Le module `Term::ReadKey` de CPAN offre une interface prête à l'emploi, et devrait être plus efficace que de lancer des `stty` pour chaque touche. Il inclut même un support limité pour Windows.

```
use Term::ReadKey;
ReadMode('cbreak');
$key = ReadKey(0);
ReadMode('normal');
```

Cependant, cela requiert que vous ayez un compilateur C fonctionnel, qui puisse être utilisé pour compiler et installer des modules de CPAN. Voici une solution n'utilisant que le module standard POSIX, qui devrait être disponible en natif sur votre système (s'il est lui-même POSIX).

```
use HotKey;
$key = readkey();
```

Et voici le module `HotKey`, qui cache les appels POSIX gérant les structures `termios`, qui sont assez ésothériques, il faut bien l'avouer :

```
HotKey.pm
package HotKey;

@ISA = qw(Exporter);
@EXPORT = qw(cbreak cooked readkey);

use strict;
use POSIX qw(:termios_h);
my ($term, $oterm, $echo, $noecho, $fd_stdin);

$fd_stdin = fileno(STDIN);
$term = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm = $term->getlflag();
```

```

$echo = ECHO | ECHOK | ICANON;
$noecho = $oterm & ~$echo;

sub cbreak {
 $term->setlflag($noecho); # ok, on ne veut pas d'écho non plus
 $term->setcc(VTIME, 1);
 $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
 $term->setlflag($oterm);
 $term->setcc(VTIME, 0);
 $term->setattr($fd_stdin, TCSANOW);
}

sub readkey {
 my $key = '';
 cbreak();
 sysread(STDIN, $key, 1);
 cooked();
 return $key;
}

END { cooked() }

1;

```

### 25.1.6 Comment vérifier si des données sont en attente depuis le clavier ?

Le moyen le plus facile pour cela est de lire une touche en mode non bloquant, en utilisant le module `Term::ReadKey` de CPAN, et en lui passant un argument de `-1` pour indiquer ce fait.

```

use Term::ReadKey;

ReadMode('cbreak');

if (defined ($char = ReadKey(-1))) {
 # il y avait un caractère disponible, maintenant dans $char
} else {
 # pas de touche pressée pour l'instant
}

ReadMode('normal'); # restaure le terminal en mode normal

```

### 25.1.7 Comment effacer l'écran ?

Si vous devez le faire de façon occasionnelle, utilisez `system` :

```
system("clear");
```

Si ce doit être une opération fréquente, sauvegardez la séquence de nettoyage pour pouvoir ensuite l'afficher 100 fois sans avoir à appeler un programme externe autant de fois :

```
$clear_string = 'clear';
print $clear_string;
```

Si vous prévoyez d'autres manipulations d'écran, comme le positionnement du curseur, etc., utilisez plutôt le module `Term::Cap` :

```
use Term::Cap;
$terminal = Term::Cap->Tgetent({OSPEED => 9600});
$clear_string = $terminal->Tputs('cl');
```

### 25.1.8 Comment obtenir la taille de l'écran ?

Si vous avez le module `Term::ReadKey` de CPAN, vous pouvez l'utiliser pour récupérer la hauteur et la largeur en caractères et en pixels:

```
use Term::ReadKey;
($wchar, $hchar, $wpixels, $hpixels) = GetTerminalSize();
```

Cela est plus portable qu'un appel direct à `ioctl`, mais n'est pas aussi illustratif :

```
require 'sys/ioctl.ph';
die "no TIOCGWINSZ " unless defined &TIOCGWINSZ;
open(TTY, "+</dev/tty") or die "No tty: $!";
unless (ioctl(TTY, &TIOCGWINSZ, $winsize='')) {
 die sprintf "$0: ioctl TIOCGWINSZ (%08x: $!)\n", &TIOCGWINSZ;
}
($row, $col, $xpixel, $ypixel) = unpack('S4', $winsize);
print "(row,col) = ($row,$col)";
print " (xpixel,ypixel) = ($xpixel,$ypixel)" if $xpixel || $ypixel;
print "\n";
```

### 25.1.9 Comment demander un mot de passe à un utilisateur ?

(Cette question n'a rien à voir avec le web. Voir une autre FAQ pour cela.)

Il y a un exemple dans `crypt` in *perlfunc*. Tout d'abord, on place le terminal en mode "sans écho", puis on lit simplement le mot de passe. Cela peut se faire par un appel à la bonne vieille fonction `ioctl()`, par l'utilisation du contrôle du terminal de POSIX (voir *POSIX* ou le Camel Book), ou encore par un appel au programme `stty`, avec divers degrés de portabilité.

On peut aussi, sur la plupart des systèmes, utiliser le module `Term::ReadKey` de CPAN, qui est le plus simple à utiliser et le plus portable en théorie.

```
use Term::ReadKey;

ReadMode('noecho');
$password = ReadLine(0);
```

### 25.1.10 Comment lire et écrire sur le port série ?

Cela dépend du système d'exploitation au-dessus duquel tourne votre programme. Dans la plupart des systèmes Unix, les ports série sont accessibles depuis des fichiers sous `/dev`; sur d'autres systèmes, les noms de périphériques seront indubitablement différents. Il y a plusieurs types de problèmes, communs à toutes les interactions avec un périphérique :

#### verrouillage

Votre système peut utiliser des fichiers de verrouillage pour contrôler les accès concurrentiels. Soyez certain de suivre le bon protocole de verrouillage. Des comportements imprévisibles peuvent survenir suite à un accès simultané, par différents processus, au même périphérique.

#### mode d'ouverture

Si vous prévoyez d'utiliser à la fois des opérations de lecture et d'écriture sur le périphérique, vous devrez l'ouvrir en mode de mise à jour (voir `open` in *perlfunc* pour plus de détails). Vous pouvez aussi vouloir l'ouvrir sans prendre le risque de bloquer, en utilisant `sysopen()` et les constantes `O_RDWR|O_NDELAY|O_NOCTTY` fournies par le module `Fcntl` (qui fait partie de la distribution standard). Se référer à `sysopen` in *perlfunc* pour plus de précisions quant à cette approche.

#### fin de ligne

Certains périphériques vont s'attendre à trouver un `"\r"` à la fin de chaque ligne plutôt qu'un `"\n"`. Sur certains portages de perl, `"\r"` et `"\n"` sont différents de leur valeurs usuelles (Unix) qui en ASCII sont respectivement `"\012"` et `"\015"`. Il se peut que vous ayez à utiliser ces valeurs numériques directement, sous leur forme octale (`"\015"`), hexadécimale (`"0x0D"`), ou sous forme de caractère de contrôle (`"\cM"`).

```
print DEV "atv1\012"; # mauvais, pour certains périphériques
print DEV "atv1\015"; # bon, pour certains périphériques
```

Bien que pour les fichiers de texte normaux, un "\n" fasse l'affaire, il n'y a toujours pas de schéma unifié pour terminer une ligne qui soit portable entre Unix, DOS/Win et Macintosh, sauf à terminer *TOUTES* les lignes par un "\015\012", et de retirer ce dont vous n'avez pas besoin dans le résultat en sortie. Cela s'applique tout particulièrement aux E/S sur les prises (sockets) et à la purge des tampons, problèmes qui sont discutés ci-après.

### purge des tampons de sortie

Si vous vous attendez à ce que tous les caractères que vous émettez avec `print()` sortent immédiatement, il vous faudra activer la purge automatique des tampons de sortie (autoflush) sur ce descripteur de fichier. Vous pouvez utiliser `select()` et la variable `$|` pour contrôler cette purge (voir `$|` in *perlvar* et `select` in *perlfunc* ou Comment vider/annuler les tampons en sortie ? Pourquoi m'en soucier ? in *perlfaq5*):

```
$oldh = select(DEV);
$| = 1;
select($oldh);
```

Vous verrez aussi du code qui réalise l'opération sans variable temporaire :

```
select((select(DEV), $| = 1)[0]);
```

Ou, si le fait de charger quelque milliers de lignes de code, simplement par peur d'une toute petite variable comme `$|`, ne vous ennuie pas outre mesure :

```
use IO::Handle;
DEV->autoflush(1);
```

Comme expliqué dans le point précédent, cela ne marchera pas si vous utilisez des E/S sur une prise entre Unix et un Macintosh. Vous devrez câbler vos terminaisons de ligne, dans ce cas.

### entrée non bloquante

Si vous effectuez une lecture bloquante par `read()` ou `sysread()`, vous devrez vous arranger pour que le déclenchement d'une alarme vous fournisse le déblocage nécessaire (voir `alarm` in *perlfunc*). Si vous avez effectué une ouverture non bloquante, vous bénéficierez certainement d'une lecture non bloquante, ce qui peut forcer l'utilisation de `select()` (dans sa version avec 4 arguments), pour déterminer si l'E/S est possible ou non sur ce périphérique (voir `select` in *perlfunc*.)

En cherchant à lire sa boîte vocale, le fameux Jamie Zawinski <jwz@netscape.com>, après de nombreux grincements de dents et une lutte avec `sysread`, `sysopen`, les caprices de la fonction POSIX `tgetatrr`, et de nombreuses autres fonctions promettant de s'éclater la nuit, est finalement arrivé à ceci :

```
sub open_modem {
 use IPC::Open2;
 my $stty = '/bin/stty -g';
 open2(*MODEM_IN, *MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
 # lancer cu trafique les paramètres de /dev/tty, même lorsqu'il a
 # été lancé depuis un tube...
 system("/bin/stty $stty");
 $_ = <MODEM_IN>;
 chomp;
 if (!m/^Connected/) {
 print STDERR "$0: cu printed '$_' instead of 'Connected'\n";
 }
}
```

### 25.1.11 Comment décoder les fichiers de mots de passe cryptés ?

En dépensant d'énormes quantités d'argent pour du matériel dédié, mais cela finira par attirer l'attention.

Sérieusement, cela n'est pas possible si ce sont des mots de passe Unix – le système de cryptage des mots de passe sur Unix utilise une fonction de hachage à sens unique. C'est plus du hachage que de l'encryption. La meilleure méthode consiste à trouver quelque chose d'autre qui se hache de la même façon. Il n'est pas possible d'inverser la fonction pour retrouver la chaîne d'origine. Des programmes comme Crack peuvent essayer de deviner les mots de passe de façon brutale (et futée), mais ne vont pas (ne peuvent pas) garantir de succès rapide.

Si vous avez peur que vos utilisateurs ne choisissent de mauvais mots de passe, vous devriez le vérifier au vol lorsqu'ils essaient de changer leur mot-de-passe (en modifiant la commande `passwd(1)` par exemple).



### 25.1.12 Comment lancer un processus en arrière plan ?

Plusieurs modules savent lancer d'autres processus sans bloquer votre programme Perl. Vous pouvez utiliser `IPC::Open3`, `Parallel::Jobs`, `IPC::Run` et quelques-uns des modules POE. Voir CPAN pour plus de détails.

Vous pouvez aussi utiliser

```
system("cmd &")
```

ou utiliser `fork` comme expliqué dans `fork` in *perlfunc*, avec des exemples supplémentaires dans *perlipc*. Voici quelques petites choses dont il vaut mieux être conscient sur un système de type Unix:

#### STDIN, STDOUT, et STDERR sont partagés.

Le processus principal et celui en arrière plan (le processus "fils") partagent les mêmes descripteurs de fichier STDIN, STDOUT et STDERR. Si les deux essaient d'y accéder en même temps, d'étranges phénomènes peuvent survenir. Il vaudrait mieux les fermer ou les réouvrir dans le fils. On peut contourner ce fait en ouvrant un tube via `open` (voir `open` in *perlfunc*) mais sur certains systèmes, cela implique que le processus fils ne puisse pas survivre à son père.

#### Signaux

Il faudra capturer les signaux SIGCHLD, et peut être SIGPIPE aussi. Un SIGCHLD est envoyé lorsque le processus en arrière plan se termine. Un SIGPIPE est envoyé lorsque l'on écrit dans un descripteur de fichier que le processus fils a fermé (ne pas capturer un SIGPIPE peut causer silencieusement la mort du processus). Cela n'est pas un problème avec `system("cmd&")`.

#### Zombies

Il faut vous préparer à "collecter" les processus fils qui se terminent :

```
$SIG{CHLD} = sub { wait };
$SIG{CHLD} = 'IGNORE';
```

Vous pouvez aussi utiliser la technique du double fork. Vous faites un `wait()` immédiat de votre fils et c'est le daemon init qui fera le `wait()` de votre petit-fils lorsqu'il se terminera.

```
unless ($pid = fork) {
 unless (fork) {
 exec "ce que vous voulez réellement faire";
 die "échec d'exec !";
 }
 exit 0;
}
waitpid($pid,0);
```

Voir Signaux in *perlipc* pour d'autres exemple de code réalisant cela. Il n'est pas possible d'obtenir des zombies avec `system("prog &")`.

### 25.1.13 Comment capturer un caractère de contrôle, un signal ?

On ne "capture" (en anglais "trap") pas vraiment un caractère de contrôle. En fait, ce caractère génère un signal qui est envoyé au groupe du terminal sur lequel tourne le processus en avant plan, signal que l'on capture ensuite dans le processus. Les signaux sont documentés dans Signaux in *perlipc* et dans le chapitre "Signaux" du Camel Book.

Vous pouvez utiliser la table de hachage %SIG pour y attacher vos fonctions de gestion des signaux. Lorsque perl reçoit un signal, il cherche dans %SIG une clé identique au nom du signal reçu et appelle la fonction associée à cette clé.

```
via un sous-programme anonyme
$SIG{INT} = sub { syswrite(STDERR, "ouch\n", 5) };

via une référence à une fonction
$SIG{INT} = \&ouch;

via une chaîne content le nom de la fonction
(référence symbolique)
$SIG{INT} = "ouch";
```

Dans les versions de Perl antérieures à 5.8, le signal était traité dès sa réception via du code C qui captait le signal et appelait directement une éventuelle fonction Perl stockée dans %SIG. Cela pouvait amener perl à planter. Depuis la version 5.8.0, perl regarde dans %SIG \*après\* la réception du signal, et non au moment de sa réception. Les versions antérieures de cette réponse étaient erronées.

### 25.1.14 Comment modifier le fichier masqué (shadow) de mots de passe sous Unix ?

Si perl a été installé correctement et que votre librairie d'accès au fichier masqué est écrite proprement, alors les fonctions `getpw*`(`)` décrites dans *perlfunc* devraient, en théorie, fournir un accès (en lecture seule) aux mots de passe masqués. Pour changer le fichier, faites une copie du fichier de masque (son format varie selon les systèmes - voir `passwd(5)` pour les détails) et utilisez `pwd_mkdb(8)` pour l'installer (voir `pwd_mkdb` pour plus de détails).

### 25.1.15 Comment positionner l'heure et la date ?

En supposant que vous tourniez avec des privilèges suffisants, vous devriez être capables de changer l'heure du système et le temps en lançant la commande `date(1)`. (Il n'y a pas moyen de positionner l'heure et la date pour un processus seulement.) Ce mécanisme fonctionnera sous Unix, MS-DOS, Windows et NT ; sous VMS une commande équivalente est `set time`.

Si par contre vous désirez seulement changer de fuseau horaire, vous pourrez certainement vous en sortir en positionnant une variable d'environnement :

```
ENV{TZ} = "MST7MDT"; # unixien
ENV{ 'SYS$TIMEZONE_DIFFERENTIAL' }="-5" # vms
system "trn comp.lang.perl.misc";
```

### 25.1.16 Comment effectuer un `sleep()` ou `alarm()` de moins d'une seconde ?

Pour obtenir une granularité plus fine que la seconde obtenue par la fonction `sleep()`, le plus simple est d'utiliser `select()` comme décrit dans `select` in *perlfunc*. Essayez aussi les modules `Time::HiRes` et `BSD::Itimer` (disponibles sur CPAN et dans la distribution standard depuis Perl 5.8 pour `Time::HiRes`).

### 25.1.17 Comment mesurer un temps inférieur à une seconde ?

En général, cela risque d'être difficile. Le module `Time::HiRes` (disponible sur CPAN et dans la distribution standard depuis Perl 5.8) apporte cette fonctionnalité sur certains systèmes.

Si votre système supporte à la fois la fonction `syscall()` en Perl et un appel système tel que `gettimeofday(2)`, alors vous pouvez peut-être faire quelque chose comme ceci :

```
require 'sys/syscall.ph';

$TIMEVAL_T = "LL";

$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_gettimeofday, $start, 0) != -1
 or die "gettimeofday: $!";

#####
FAITES VOS OPERATIONS ICI
#####

syscall(&SYS_gettimeofday, $done, 0) != -1
 or die "gettimeofday: $!";

@start = unpack($TIMEVAL_T, $start);
@done = unpack($TIMEVAL_T, $done);

corriger les microsecondes
for ($done[1], $start[1]) { $_ /= 1_000_000 }

$delta_time = sprintf "%.4f", ($done[0] + $done[1])
-
($start[0] + $start[1]);
```

### 25.1.18 Comment réaliser un `atexit()` ou `setjmp()/longjmp()` ? (traitement d'exceptions)

La version 5 de Perl apporte le bloc `END`, qui peut être utilisé pour simuler `atexit()`. Le bloc `END` de chaque paquetage est appelé lorsque le programme ou le fil d'exécution se termine (voir la page *perlmod* pour plus de détails).

Par exemple, on peut utiliser ceci pour s'assurer qu'un programme filtre est bien parvenu à vider son tampon de sortie sans remplir tout le disque :

```
END {
 close(STDOUT) || die "stdout close failed: $!";
}
```

Par contre, le bloc `END` n'est pas appelé lorsqu'un signal non capturé tue le programme, donc si vous utilisez les blocs `END`, vous devriez aussi utiliser :

```
use sigtrap qw(die normal-signals);
```

En Perl, le traitement des exceptions s'effectue au travers de l'opération `eval()`. Vous pouvez utiliser `eval()` en lieu et place de `setjmp`, et `die()` pour `longjmp()`. Pour plus de détails sur cela, lire la section sur les signaux, et plus particulièrement le traitement limitant le temps de blocage pour un `flock()` dans Signaux in *perlipc* et le chapitre "Signaux" du Camel Book.

Si tout ce qui vous intéresse est le traitement des exceptions proprement dit, essayez la bibliothèque `exceptions.pl` (qui fait partie de la distribution standard de Perl).

Si vous préférez la syntaxe de `atexit()` (et désirez `rmexit()` aussi), essayez le module `AtExit` disponible sur CPAN.

### 25.1.19 Pourquoi mes programmes avec `socket()` ne marchent pas sous System V (Solaris) ? Que signifie le message d'erreur « Protocole non supporté » ?

Certains systèmes basés sur Sys-V, et particulièrement Solaris 2.X, ont redéfini certaines constantes liée aux prises (`sockets`) et qui étaient des standards de fait. Étant donné que ces constantes étaient communes sur toutes les architectures, elles étaient souvent câblées dans le code perl. La bonne manière de résoudre ce problème est d'utiliser "use Socket" pour obtenir les valeurs correctes.

Notez que bien que SunOS et Solaris soient compatibles au niveau des binaires, ces valeurs sont néanmoins différentes. Allez comprendre !

### 25.1.20 Comment appeler les fonctions C spécifiques à mon système depuis Perl ?

Dans la plupart des cas, on écrit un module externe – voir la réponse à la question "Où puis-je apprendre à lier du C avec Perl? [h2xs, xsubpp]". Cependant, si la fonction est un appel système et que votre système supporte `syscall()`, vous pouvez l'utiliser pour ce faire (documentée dans *perlfunc*).

Rappelez-vous de regarder les modules qui sont livrés avec votre distribution, ainsi que ceux disponibles sur CPAN - quel qu'un a peut-être déjà écrit un module pour le faire. Sur Windows, essayez `Win32::API`. Sur Mac, essayez `Mac::Carbon`. Si aucun module ne propose d'interface vers cette fonction C, vous pouvez insérer un peu de code C directement dans votre code Perl via le module `Inline::C`.

### 25.1.21 Où trouver les fichiers d'inclusion pour `ioctl()` et `syscall()` ?

Historiquement, ceux-ci sont générés par l'outil `h2ph`, inclus dans les distributions standard de perl. Ce programme convertit les directives `cpp(1)` du fichier d'inclusion C en un fichier contenant des définitions de sous-routines, comme `&SYS_getitimer`, qui peuvent ensuite être utilisées comme argument de vos fonctions. Cela ne fonctionne pas parfaitement, mais l'essentiel du travail est fait. Des fichiers simples comme *errno.h*, *syscall.h*, et *socket.h* donnent de bons résultats, mais de plus complexes comme *ioctl.h* demandent presque toujours une intervention manuelle après coup. Voici comment installer les fichiers \*.ph:

1. devenir super-utilisateur
2. `cd /usr/include`
3. `h2ph *.h */*.h`

Si votre système supporte le chargement dynamique, et pour des raisons de portabilité et de santé, vous devriez plutôt utiliser `h2xs` (qui fait lui aussi partie de la distribution standard de perl). Cet outil convertit un fichier d'inclusion C en une extension Perl. Voir *perlxstut* pour savoir comment débiter avec `h2xs`.

Si votre système ne supporte pas le chargement dynamique, vous pouvez néanmoins utiliser `h2xs`. Voir *perlxstut* et *ExtUtils::MakeMaker* pour plus d'information (brièvement, utilisez simplement **make perl** et non un simple **make** pour reconstruire un perl avec une nouvelle extension statiquement liée).

### 25.1.22 Pourquoi les scripts perl en setuid se plaignent-ils d'un problème noyau ?

Certains systèmes d'exploitation ont un bug dans leur noyau qui rend les scripts setuid [NDT : fichiers dont le bit 's' est positionné sur l'exécutable, par exemple avec "chmod u+s" sous Unix] non sûrs intrinsèquement. Perl vous offre quelques options (décrites dans *perlsec*) pour contourner ce fait sur ces systèmes.

### 25.1.23 Comment ouvrir un tube depuis et vers une commande simultanément ?

Le module IPC::Open2 (qui fait partie de la distribution perl standard) est une approche aisée qui utilise en interne les appels systèmes pipe(), fork() et exec(). Cependant, soyez sûrs de bien lire les avertissements concernant les interblocages dans sa documentation (voir *IPC::Open2*). Voir aussi Communication bidirectionnelle avec un autre processus in *perlipc* et Communication bidirectionnelle avec vous-même in *perlipc*.

On peut aussi utiliser le module IPC::Open3 (lui aussi dans la distribution standard), mais attention: l'ordre des arguments est différent de celui utilisé par IPC::Open2 (voir *IPC::Open3*).

### 25.1.24 Pourquoi ne puis-je pas obtenir la sortie d'une commande avec system() ?

Vous confondez system() avec les apostrophes inversées (backticks, ``). La fonction system() lance une commande et retourne sa valeur de sortie (une valeur sur 16 bits : les 7 bits de poids faible indiquent le signal qui a tué le processus le cas échéant, et les 8 bits de poids fort sont la valeur de sortie effective). Les apostrophes inversées (``) lancent une commande et retournent ce que cette commande a émis sur STDOUT.

```
$exit_status = system("mail-users");
$output_string = `ls`;
```

### 25.1.25 Comment capturer la sortie STDERR d'une commande externe ?

Il y a trois moyens fondamentaux de lancer une commande externe :

```
system $cmd; # avec system()
$output = ` $cmd `; # avec les apostrophes inversées (``)
open (PIPE, "cmd |"); # avec open()
```

Avec system(), STDOUT et STDERR vont tous deux aller là où ces descripteurs sont dirigés dans le script lui-même, sauf si la commande system() les redirige explicitement par ailleurs. Les apostrophes inversées et open() lisent **uniquement** la sortie (STDOUT) de votre commande externe.

Vous pouvez aussi utiliser la fonction open3() du module IPC::Open3. Benjamin Goldberg propose les quelques exemples de code suivants.

Pour récupérer le STDOUT de votre programme externe et se débarrasser de son STDERR :

```
use IPC::Open3;
use File::Spec;
use Symbol qw(gensym);
open(NULL, ">", File::Spec->devnull);
my $pid = open3(gensym, *PH, ">&NULL", "cmd");
while(<PH>) { }
waitpid($pid, 0);
```

Pour récupérer le STDERR de votre programme externe et se débarrasser de son STDOUT :

```
use IPC::Open3;
use File::Spec;
use Symbol qw(gensym);
open(NULL, ">", File::Spec->devnull);
my $pid = open3(gensym, ">&NULL", *PH, "cmd");
while(<PH>) { }
waitpid($pid, 0);
```

Pour récupérer le STDERR de votre programme externe, et rediriger son STDOUT vers votre propre STDERR :

```
use IPC::Open3;
use Symbol qw(gensym);
my $pid = open3(gensym, ">&STDERR", *PH, "cmd");
while(<PH>) { }
waitpid($pid, 0);
```

Pour récupérer séparément le STDOUT et le STDERR de votre commande externe, vous pouvez les rediriger vers des fichiers temporaires, exécuter la commande puis lire les fichiers temporaires :

```
use IPC::Open3;
use Symbol qw(gensym);
use IO::File;
local *CATCHOUT = IO::File->new_tmpfile;
local *CATCHERR = IO::File->new_tmpfile;
my $pid = open3(gensym, ">&CATCHOUT", ">&CATCHERR", "cmd");
waitpid($pid, 0);
seek $_, 0, 0 for *CATCHOUT, *CATCHERR;
while(<CATCHOUT>) {}
while(<CATCHERR>) {}
```

Mais les *\*deux\** fichiers temporaires ne sont pas absolument indispensables. Le code suivant fonctionnera aussi bien, sans interblocage :

```
use IPC::Open3;
use Symbol qw(gensym);
use IO::File;
local *CATCHERR = IO::File->new_tmpfile;
my $pid = open3(gensym, *CATCHOUT, ">&CATCHERR", "cmd");
while(<CATCHOUT>) {}
waitpid($pid, 0);
seek CATCHERR, 0, 0;
while(<CATCHERR>) {}
```

Et il sera aussi plus rapide puisque vous pourrez traiter la sortie du programme immédiatement plutôt que d'attendre la terminaison du programme.

Avec toutes ces routines, vous pouvez changer les descripteurs de fichier avant l'appel :

```
open(STDOUT, ">logfile");
system("ls");
```

ou vous pouvez utiliser les redirections du shell de Bourne :

```
$output = `cmd 2>some_file`;
open (PIPE, "cmd 2>some_file |");
```

Vous pouvez aussi utiliser les redirections de fichier pour rendre STDERR un synonyme de STDOUT :

```
$output = `cmd 2>&1`;
open (PIPE, "cmd 2>&1 |");
```

Cependant, vous ne *pouvez pas* simplement ouvrir STDERR en synonyme de STDOUT depuis votre programme Perl, et ne pas recourir ensuite à la redirection en shell. Ceci ne marche pas :

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`; # stderr n'est toujours pas capturé
```

Cela échoue parce que `open()` rend `STDERR` un synonyme de `STDOUT` au moment où l'appel à `open()` a été effectué. Les apostrophes inversées redirigent ensuite `STDOUT` vers une chaîne, mais ne changent pas `STDERR` (qui va toujours là où allait le `STDOUT` d'origine).

Notez bien que vous *devez* utiliser la syntaxe de redirection du shell de Bourne (`sh(1)`), et non celle de `csh(1)` ! Des précisions sur les raisons pour lesquelles Perl utilise le shell de Bourne pour `system()` et les apostrophes inversées, ainsi que lors des ouvertures de tubes, se trouvent dans l'article *versus/csh.whynot* de la collection "Far More Than You Ever Wanted To Know" sur <http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz>.

Pour capturer à la fois le `STDOUT` et le `STDERR` d'une commande :

```
$output = 'cmd 2>&1'; # soit avec des apostrophes
$pid = open(PH, "cmd 2>&1 |"); # inversées, soit avec un tube
while (<PH>) { } # plus une lecture
```

Pour capturer seulement le `STDOUT` et jeter le `STDERR` d'une commande :

```
$output = 'cmd 2>/dev/null'; # soit avec des apostrophes
$pid = open(PH, "cmd 2>/dev/null |"); # inversées, soit avec un tube
while (<PH>) { } # plus une lecture
```

Pour capturer seulement le `STDERR` et jeter le `STDOUT` d'une commande :

```
$output = 'cmd 2>&1 1>/dev/null'; # soit avec des apostrophes
$pid = open(PH, "cmd 2>&1 1>/dev/null |"); # inversées, soit avec un tube
while (<PH>) { } # plus une lecture
```

Pour échanger le `STDOUT` et le `STDERR` d'une commande afin d'en capturer le `STDERR` mais laisser le `STDOUT` sortir à la place du `STDERR` d'origine :

```
$output = 'cmd 3>&1 1>&2 2>&3 3>&-'; # soit avec des apostrophes
$pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|"); # inversées, soit avec un tube
while (<PH>) { } # plus une lecture
```

Pour lire séparément le `STDOUT` et le `STDERR` d'une commande, il est plus facile de les rediriger chacun dans un fichier, et ensuite de les lire lorsque la commande est terminée :

```
system("program args 1>program.stdout 2>program.stderr");
```

L'ordre est important dans tout ces exemples. Cela est dû au fait que le shell prend toujours en compte les redirections en les lisant strictement de gauche à droite.

```
system("prog args 1>tmpfile 2>&1");
system("prog args 2>&1 1>tmpfile");
```

La première commande redirige à la fois la sortie standard et la sortie d'erreur dans le fichier temporaire. La seconde commande n'y envoie que la sortie standard d'origine, et la sortie d'erreur d'origine se retrouve envoyé vers la sortie standard d'origine.

### 25.1.26 Pourquoi `open()` ne retourne-t-il pas d'erreur lorsque l'ouverture du tube échoue ?

Si le second argument d'un `open()` utilisant un tube contient des métacaractères du shell, perl fait un `fork()` puis exécute un shell pour décoder les métacaractères et éventuellement exécuter le programme voulu. Si le programme ne peut s'exécuter, c'est le shell qui reçoit le message et non Perl. Tout ce que verra Perl c'est s'il a réussi à lancer le shell. Vous pouvez toujours capturer la sortie d'erreur `STDERR` du shell et y chercher les messages d'erreur. Voir "Comment capturer la sortie de `STDERR` d'une commande externe ?" ou utiliser `IPC::Open3`.

S'il n'y a pas de métacaractères dans l'argument de `open()`, Perl exécute la commande directement, sans utiliser le shell, et peut donc correctement rapporter la bonne exécution de la commande.

### 25.1.27 L'utilisation des apostrophes inversées dans un contexte vide pose-t-elle problème ?

À strictement parler, non. Par contre, stylistiquement parlant, ce n'est pas un bon moyen d'écrire du code maintenable. Perl propose plusieurs opérateurs permettant l'exécution de commandes externes. Les apostrophes inversées en sont un : elles capturent la sortie produite par la commande pour que vous l'utilisiez dans votre programme. La fonction `system` en est un autre : elle ne fait pas cette capture.

Considérons la ligne suivante :

```
'cat /etc/termcap';
```

On a oublié de vérifier la valeur de  `$?`  pour savoir si le programme s'était bien déroulé. Même si l'on avait écrit :

```
print 'cat /etc/termcap';
```

ce code pourrait et devrait probablement être écrit ainsi :

```
system("cat /etc/termcap") == 0
 or die "cat program failed!";
```

Ce qui permet d'avoir la sortie rapidement (au fur et à mesure de sa génération, au lieu d'avoir à attendre jusqu'à la fin) et vérifie aussi la valeur de sortie.

Avec `system()` vous avez aussi un contrôle direct sur une possible interprétation de méta-caractères, alors que ce n'est pas permis avec des apostrophes inversées.

### 25.1.28 Comment utiliser des apostrophes inversées sans traitement du shell ?

C'est un peu tordu. Vous ne pouvez pas simplement écrire :

```
@ok = 'grep @opts '$search_string' @filenames';
```

Depuis Perl 5.8.0, vous pouvez utiliser `open()` avec sa syntaxe à plusieurs arguments. De la même manière que pour les appels avec plusieurs arguments des fonctions `system()` et `exec()`, il n'y a alors plus de passage pas le shell.

```
open(GREP, "-|", 'grep', @opts, $search_string, @filenames);
chomp(@ok = <GREP>);
close GREP;
```

Vous pouvez aussi faire :

```
my @ok = ();
if (open(GREP, "-|")) {
 while (<GREP>) {
 chomp;
 push(@ok, $_);
 }
 close GREP;
} else {
 exec 'grep', @opts, $search_string, @filenames;
}
```

De même qu'avec `system()`, il n'y a aucune intervention du shell lorsqu'on donne une liste d'arguments à `exec()`. D'autres exemples de ceci se trouvent dans Ouvertures Sûres d'un Tube in *perlipc*.

Notez que si vous utilisez Microsoft, aucune solution à ce problème vexant n'est possible. Même si Perl pouvait émuler `fork()`, vous seriez toujours coincés, parce que Microsoft ne fournit aucune API du style `argc/argv`.

### 25.1.29 Pourquoi mon script ne lit-il plus rien de STDIN après que je lui ai envoyé EOF (^D sur Unix, ^Z sur MS-DOS) ?

Certaines implémentations de stdio positionnent des indicateurs d'erreur et de fin de fichier qui demandent à être annulées d'abord. Le module POSIX définit `clearerr()` que vous pouvez utiliser. C'est la façon correcte de traiter ce problème. Voici d'autres alternatives, moins fiables :

1. Gardez sous la main la position dans le fichier, et retournez-y, ainsi :
 

```
$where = tell(LOG);
seek(LOG, $where, 0);
```
2. Si cela ne marche pas, essayez de retourner à d'autres parties du fichier, et puis finalement là où vous le voulez.
3. Si cela ne marche toujours pas, essayez d'aller ailleurs dans le fichier, de lire quelque chose, puis de retourner à l'endroit mémorisé.
4. Si cela ne marche toujours pas, abandonnez votre librairie stdio et utilisez `sysread` directement.

### 25.1.30 Comment convertir mon script shell en perl ?

Apprenez Perl et réécrivez le. Sérieusement, il n'y a pas de convertisseur simple. Des choses compliquées en shell sont simples en Perl, et cette complexité même rendrait l'écriture d'un convertisseur shell->perl quasi impossible. En réécrivant le programme, vous penserez plus à ce que vous désirez réellement faire, et vous vous affranchirez du paradigme de flots de données mis bout à bout propre au shell, qui, bien que pratique pour certains problèmes, est à l'origine de nombreuses inefficacités.

### 25.1.31 Puis-je utiliser perl pour lancer une session telnet ou ftp ?

Essayez les modules `Net::FTP`, `TCP::Client` et `Net::Telnet` (disponibles sur CPAN). Pour émuler le protocole telnet, on s'aidera de <http://www.cpan.org/scripts/netstuff/telnet.emul.shar>, mais il est probablement plus facile d'utiliser `Net::Telnet` directement.

Si tout ce que vous voulez faire est prétendre d'être telnet mais n'avez pas du tout besoin de la négociation initiale, alors l'approche classique bi-processus suffira :

```
use IO::Socket; # nouveau dans 5.004
$handle = IO::Socket::INET->new('www.perl.com:80')
 || die "can't connect to port 80 on www.perl.com: $!";
$handle->autoflush(1);
if (fork()) { # XXX: undef signifie un échec
 select($handle);
 print while <STDIN>; # tout ce qui vient de stdin va vers la prise
} else {
 print while <$handle>; # tout ce qui vient de la prise va vers stdout
}
close $handle;
exit;
```

### 25.1.32 Comment écrire "expect" en Perl ?

Il était une fois une librairie appelée `chat2.pl` (incluse dans la distribution standard de perl) qui ne fut jamais vraiment terminée. Si vous la trouvez quelque part, *ne l'utilisez pas*. De nos jours, il est plus rentable de regarder du côté du module `Expect`, disponible sur CPAN, qui requiert aussi deux autres modules de CPAN, `IO::Pty` and `IO::Stty`.

### 25.1.33 Peut-on cacher les arguments de perl sur la ligne de commande aux programmes comme "ps" ?

Tout d'abord, notez que si vous faites cela pour des raisons de sécurité (afin d'empêcher les autres de voir des mots de passe, par exemple), alors vous feriez mieux de réécrire votre programme de façon à assurer que les données critiques ne sont jamais passées comme argument. Cacher ces arguments ne rendra jamais votre programme complètement sûr.

Pour vraiment altérer la ligne de commande visible, on peut assigner quelque chose à la variable `$0`, ainsi que documenté dans *perlvar*. Cependant, cela ne marchera pas sur tous les systèmes d'exploitation. Des logiciels démons comme `sendmail` y placent leur état, comme dans :

```
$0 = "orcus [accepting connections]";
```



### 25.1.34 J'ai {changé de répertoire, modifié mon environnement} dans un script perl. Pourquoi les changements disparaissent-ils lorsque le script se termine ? Comment rendre mes changements visibles ?

#### Unix

À proprement parler, ce n'est pas possible – le script s'exécute dans un processus différent du shell qui l'a démarré. Les changements effectués dans ce processus ne sont pas transmis à son parent – seulement à ses propres enfants créés après le changement en question. Il y a cependant une astuce du shell qui peut permettre de le simuler en `eval()`uant la sortie du script dans votre shell ; voir la FAQ de `comp.unix.questions` pour plus de détails.

### 25.1.35 Comment fermer le descripteur de fichier attaché à un processus sans attendre que ce dernier se termine ?

En supposant que votre système autorise ce genre de chose, il suffit d'envoyer un signal approprié à ce processus (voir `kill` in *perlfunc*). Il est d'usage d'envoyer d'abord un signal `TERM`, d'attendre un peu, et ensuite seulement d'envoyer le signal `KILL` pour y mettre fin.

### 25.1.36 Comment lancer un processus démon ?

Si par processus démon vous entendez un processus qui s'est détaché (dissocié de son terminal de contrôle), alors le mode opératoire suivant marche en général sur la plupart des systèmes Unix. Les utilisateurs de plate-forme non-Unix doivent regarder du côté du module `Votre_OS::Process` pour des solutions alternatives.

- Ouvrir `/dev/tty` et utiliser l'ioctl `TIOCNOTTY` dessus. Voir *tty* pour les détails. Ou mieux encore, utiliser la fonction `POSIX::setsid()`, pour ne pas avoir à se soucier des groupes de processus.
- Positionner le répertoire courant à `/`
- Re-ouvrir `STDIN`, `STDOUT` et `STDERR` pour qu'ils ne soient plus attachés à leur terminal d'origine.
- Se placer en arrière plan ainsi :
 

```
fork && exit;
```

Le module `Proc::Daemon`, disponible sur le CPAN, fournit une fonction qui réalise ces actions pour vous.

### 25.1.37 Comment savoir si je tourne de façon interactive ou pas ?

Bonne question. Parfois `-t STDIN` et `-t STDOUT` peuvent donner quelque indice, parfois non.

```
if (-t STDIN && -t STDOUT) {
 print "Now what? ";
}
```

Sur les systèmes POSIX, on peut tester son groupe de processus pour voir s'il correspond au groupe du terminal de contrôle comme suit :

```
use POSIX qw/getpgrp tcgetpgrp/;
open(TTY, "/dev/tty") or die $!;
$tpgrp = tcgetpgrp(fileno(*TTY));
$pgrp = getpgrp();
if ($tpgrp == $pgrp) {
 print "foreground\n";
} else {
 print "background\n";
}
```

### 25.1.38 Comment sortir d'un blocage sur événement lent ?

Utiliser la fonction `alarm()`, probablement avec le recours à une capture de signal, comme documenté dans *Signaux* in *perlipc* et le chapitre "Signaux" du Camel Book. On peut aussi utiliser le module plus flexible `Sys::AlarmCall`, disponible sur CPAN.

La fonction `alarm()` n'est pas disponible dans certaines versions de Windows. Pour le savoir, cherchez dans la documentation de votre distribution Perl.

### 25.1.39 Comment limiter le temps CPU ?

Utiliser le module BSD::Resource de CPAN.

### 25.1.40 Comment éviter les processus zombies sur un système Unix ?

Utiliser le collecteur de Signaux in *perlipc* pour appeler `wait()` lorsque le signal SIGCHLD est reçu, ou bien utiliser la technique du double `fork()` décrite dans Comment lancer un processus en arrière plan ? in *perlfaq8*.

### 25.1.41 Comment utiliser une base de données SQL ?

Le module DBI fournit une interface générique vers la plupart des serveurs ou des systèmes de base de données tels que Oracle, DB2, Sybase, mysql, Postgresql, ODBC ou même des fichiers classiques. Le module DBI accède à ces différents types de base de données à travers un pilote spécifique à chaque base de données, ou DBD (Data Base Driver). Vous pouvez obtenir la liste complète des pilotes disponibles sur CPAN en consultant <http://www.cpan.org/modules/by-module/DBD/>. Lisez <http://dbi.perl.org> pour en apprendre plus concernant DBI.

D'autres modules donnent accès à des bases spécifiques : Win32::ODBC, Alzabo, iodbc et d'autres encore trouvables en cherchant sur CPAN : <http://search.cpan.org>.

### 25.1.42 Comment terminer un appel à `system()` avec un `control-C` ?

Ce n'est pas possible. Vous devez imiter l'appel à `system()` vous-même (voir *perlipc* pour un exemple de code) et ensuite fournir votre propre routine de traitement pour le signal INT qui passera le signal au sous-processus. Ou vous pouvez vérifier si ce signal a été reçu :

```
$rc = system($cmd);
if ($rc & 127) { die "signal death" }
```

### 25.1.43 Comment ouvrir un fichier sans bloquer ?

Si vous avez assez de chance pour utiliser un système supportant les lectures non bloquantes (ce qui est le cas de la plupart des systèmes Unix), vous devez simplement utiliser les attributs `O_NDELAY` ou `O_NONBLOCK` du module `Fcntl` en les spécifiant à `sysopen()` :

```
use Fcntl;
sysopen(FH, "/foo/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
or die "can't open /foo/somefile: $!";
```

### 25.1.44 Comment faire la différence entre les erreurs shell et les erreurs perl ?

(contribution de brian d foy, <bdfoy@cpan.org>)

Lorsque vous lancez un script Perl, il y a quelque chose qui exécute le script pour vous et ce quelque chose peut émettre des messages d'erreur. Le script lui-même peut émettre ses propres messages d'avertissement et d'erreur. La plupart du temps, vous ne pouvez pas savoir lequel a émis quoi.

Vous ne pouvez sans doute pas modifier la chose qui exécute perl, mais vous pouvez modifier la manière dont perl affiche ses avertissements et ses erreurs en définissant vos propres fonctions de gestion de ces messages.

Considérons le script suivant, qui contient une erreur pas visiblement évidente :

```
#!/usr/local/bin/perl

print "Hello World\n";
```

Lorsque je tente d'exécuter ce script depuis mon shell (il se trouve que c'est bash), j'obtiens une erreur. Il semblerait que perl à oublié sa fonction `print()` mais en fait c'est ma ligne de shebang (`#!...`) qui ne contient pas le chemin d'accès à perl. Donc le shell lance le script et j'obtiens l'erreur :

```
$./test
./test: line 3: print: command not found
```

Une correction sale et rapide nécessite un petit peu plus de code mais vous aidera à localiser l'origine du problème.

```
#!/usr/bin/perl -w

BEGIN {
 $SIG{__WARN__} = sub{ print STDERR "Perl: ", @_; };
 $SIG{__DIE__} = sub{ print STDERR "Perl: ", @_; exit 1};
}

$a = 1 + undef;
$x / 0;
__END__
```

Tous les messages de perl seront maintenant préfixés par "Perl: ". Le bloc BEGIN fonctionne dès la compilation et donc tous les messages d'erreur et d'avertissement du compilateur seront eux aussi préfixés par "Perl: ".

```
Perl: Useless use of division (/) in void context at ./test line 9.
Perl: Name "main::a" used only once: possible typo at ./test line 8.
Perl: Name "main::x" used only once: possible typo at ./test line 9.
Perl: Use of uninitialized value in addition (+) at ./test line 8.
Perl: Use of uninitialized value in division (/) at ./test line 9.
Perl: Illegal division by zero at ./test line 9.
Perl: Illegal division by zero at -e line 3.
```

Si vous ne voyez pas "Perl: " devant le message, c'est qu'il ne vient pas de perl.

Vous pourriez aussi tout simplement connaître toutes les erreurs de perl mais, bien que ce soit effectivement le cas de certaines personnes, ce n'est certainement pas le vôtre. En revanche, elles sont toutes répertoriées dans *perldiag*. Donc si vous n'y trouvez pas cette erreur alors ce que ce n'est probablement pas une erreur perl.

Chercher parmi tous les messages d'erreur n'est pas facile alors laissez donc perl le faire pour vous. Utilisez la directive `use diagnostics;` qui transforme les messages d'erreurs normaux de perl en un discours un peu plus long sur le sujet.

```
use diagnostics;
```

Si vous n'obtenez pas un ou deux paragraphes d'explication, il y a de grande chance que ce ne soit pas une erreur perl.

### 25.1.45 Comment installer un module du CPAN ?

La façon la plus simple est d'utiliser un module lui aussi appelé CPAN qui le fera pour vous. Ce module est distribué en standard avec les versions de perl 5.004 ou plus.

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan> install Some::Module
```

Pour installer manuellement le module CPAN, ou tout autre module de CPAN qui se comporte normalement, suivez les étapes suivantes :

1. Désarchiver les sources dans une zone temporaire.
2. `perl Makefile.PL`
3. `make`
4. `make test`
5. `make install`

Si votre version de perl est compilée sans support pour le chargement dynamique de bibliothèques, alors il vous faudra remplacer l'étape 3 (**make**) par **make perl** et vous obtiendrez un nouvel exécutable *perl* avec votre extension liée statiquement.

Voir *ExtUtils::MakeMaker* pour plus de détails sur les extensions de fabrication. Voir aussi la question suivante, Quelle est la différence entre `require` et `use` ? (§25.1.46).

### 25.1.46 Quelle est la différence entre `require` et `use` ?

Perl offre différentes solutions pour inclure du code d'un fichier dans un autre. Voici les différences entre les quelques constructions disponibles pour l'inclusion :

- 1) "`do $file`" ressemble à "`eval 'cat $file'`", sauf que le premier
  - 1.1: cherche dans `@INC` et met à jour `%INC`.
  - 1.2: entoure le code `eval()`ué d'une portée lexicale *\*indépendante\**.
- 2) "`require $file`" ressemble à "`do $file`", sauf que le premier
  - 2.1: s'arrange pour éviter de charger deux fois un même fichier.
  - 2.2: lance une exception si la recherche, la compilation ou l'exécution de `$file` échoue.
- 3) "`require Module`" ressemble à "`require 'Module.pm'`", sauf que le premier
  - 3.1: traduit chaque `::` en votre séparateur de répertoire.
  - 3.2: indique au compilateur que `Module` est une classe, passible d'appels indirects.
- 4) "`use Module`" ressemble à "`require Module`", sauf que le premier
  - 4.1: charge le module à la phase de compilation, et non à l'exécution.
  - 4.2: importe ses symboles et sémantiques dans la paquetage courant.

En général, on utilise `use` avec un module Perl adéquat.

### 25.1.47 Comment gérer mon propre répertoire de modules/bibliothèques ?

Lorsque vous fabriquez les modules, utilisez les options `PREFIX` et `LIB` à la phase de génération de `Makefiles` :

```
perl Makefile.PL PREFIX=/mydir/perl LIB=/mydir/perl/lib
```

puis, ou bien positionnez la variable d'environnement `PERL5LIB` avant de lancer les scripts utilisant ces modules/bibliothèques (voir *perlrun*), ou bien utilisez :

```
use lib '/mydir/perl/lib';
```

C'est presque la même chose que :

```
BEGIN {
 unshift(@INC, '/mydir/perl/lib');
}
```

sauf que le module `lib` vérifie les sous-répertoires dépendants de la machine. Voir le pragma *lib* de Perl pour plus d'information.

### 25.1.48 Comment ajouter le répertoire dans lequel se trouve mon programme dans le chemin de recherche des modules / bibliothèques ?

```
use FindBin;
use lib "$FindBin::Bin";
use vos_propres_modules;
```

### 25.1.49 Comment ajouter un répertoire dans mon chemin de recherche (`@INC`) à l'exécution ?

Voici les moyens suggérés de modifier votre chemin de recherche :

```
la variable d'environnement PERLLIB
la variable d'environnement PERL5LIB
l'option perl -Idir sur la ligne de commande
le pragma use lib, comme dans
 use lib "$ENV{HOME}/ma_propre_biblio_perl";
```

Ce dernier est particulièrement utile, parce qu'il prend en compte les fichiers propres à une architecture donnée. Le module pragmatique `lib.pm` a été introduit dans la version 5.002 de Perl.

### 25.1.50 Qu'est-ce que socket.ph et où l'obtenir ?

C'est un fichier à la mode de perl4 définissant des constantes pour la couche réseau du système. Il est parfois construit en utilisant h2ph lorsque Perl est installé, mais d'autres fois il ne l'est pas. Les programmes modernes utilisent use Socket ; à la place.

## 25.2 AUTEURS

Copyright (c) 1997-1999 Tom Christiansen, Nathan Torkington et d'autres auteurs sus-cités. Tous droits réservés.

Cette documentation est libre ; vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même. Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code en précisant l'origine serait de bonne courtoisie mais n'est pas obligatoire.

## 25.3 TRADUCTION

### 25.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 25.3.2 Traducteur

Traduction initiale : Raphaël Manfredi <Raphael\_Manfredi@grenoble.hp.com>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit (paul.gaborit at enstimac.fr).

### 25.3.3 Relecture

Simon Washbrook, Gérard Delafond.

# Chapitre 26

## perlfaq9

Réseau

### 26.1 DESCRIPTION

Cette section traite des questions relatives aux aspects réseau, à internet et un peu au web.

#### 26.1.1 Quelle est la forme correcte d'une réponse d'un script CGI ?

(Alan Flavell <flavell+www@a5.ph.gla.ac.uk> répond...)

La Common Gateway Interface (CGI) spécifie une interface logicielle entre un programme ("le script CGI") et un serveur web (HTTPD). Cette interface n'est pas spécifique à Perl et possède ses propres FAQ et tutoriels ainsi qu'un forum de discussions [comp.infosystems.www.authoring.cgi](http://comp.infosystems.www.authoring.cgi).

Les spécifications CGI sont présentées dans la RFC d'information 3875 (<http://www.ietf.org/rfc/rfc3875>).

D'autres documentations intéressantes sont listées sur [http://www.perl.org/CGI\\_MetaFAQ.html](http://www.perl.org/CGI_MetaFAQ.html).

Ces FAQ Perl répondent à quelques problèmes CGI. Mais nous incitons fortement les programmeurs Perl à utiliser le module CGI.pm qui fait attention à tous les détails pour eux.

La ressemblance entre les en-têtes d'une réponse CGI (définis dans les spécifications CGI) et ceux d'une réponse HTTP (tels que défini dans les spécifications HTTP, RFC2616) sont volontaire mais peut parfois amener à certaines confusions.

Les spécifications CGI définissent deux sortes de scripts : les scripts "Parsed Header" (dont les en-têtes sont traités) et les scripts "Non Parsed Header" ou NPH (dont les en-têtes ne sont pas traités). Chercher dans la documentation de votre serveur pour savoir ce qu'il accepte. Les scripts "Parsed Header" sont plus simples pour plusieurs raisons. Les spécifications CGI autorisent n'importe quelle représentation usuelle des fins de ligne dans la réponse CGI (c'est au serveur HTTP de produire à partir de cette réponse, une réponse HTTP correcte). Donc un "\n" écrit en mode texte est techniquement correct et recommandé. Les scripts NPH sont plus exigeants : ils doivent produire une réponse contenant un ensemble d'en-têtes HTTP complets et corrects. Or les spécifications HTTP exigent des lignes terminées par un retour chariot (carriage-return) puis un saut de ligne (line-feed), c'est à dire les valeurs ASCII \015 et \012 écrites en mode binaire.

L'utilisation de CGI.pm permet d'être indépendant de la plateforme, même si celle-ci est un système EBCDIC. CGI.pm choisit pour les fins de ligne la représentation appropriée (\$CGI::CRLF) et positionne le mode binaire comme il faut.

#### 26.1.2 Mon script CGI fonctionne en ligne de commandes mais pas depuis un navigateur. (500 Server Error)

Plusieurs problèmes peuvent en être la cause. Consultez le guide "Troubleshooting Perl CGI scripts" ("Problème de scripts CGI en Perl") sur :

<http://www.perl.org/troubleshooting CGI.html>

Ensuite, si vous pouvez montrer que vous avez lu les FAQ et que votre problème n'est pas quelque chose de simple dont on trouve facilement la réponse, vous recevrez sans doute des réponses courtoises et utiles en postant votre question sur `comp.infosystems.www.authoring.cgi` (si elle a un quelconque rapport avec les protocoles HTTP et CGI). Les questions qui semblent porter sur Perl mais qui en fait sont liées à CGI et qui sont postées sur `comp.lang.perl.misc` sont souvent mal accueillies.

Les FAQ utiles et les guides d'aide sont listés dans la Meta FAQ CGI :

```
http://www.perl.org/CGI_MetaFAQ.html
```

### 26.1.3 Comment faire pour obtenir de meilleurs messages d'erreur d'un programme CGI ?

Utilisez le module `CGI::Carp`. Il remplace `warn` et `die`, plus les fonctions normales `carp`, `croak`, et `confess` du module `Carp` qui sont des versions plus parlantes et sûres. Les erreurs continuent à être envoyées vers le fichier normal des erreurs du serveur.

```
use CGI::Carp;
warn "This is a complaint";
die "But this one is serious";
```

L'utilisation suivante de `CGI::Carp` redirige également les erreurs vers un fichier de votre choix, mais aussi les avertissements durant la phase de compilation en étant placé dans un bloc `BEGIN`.

```
BEGIN {
 use CGI::Carp qw(carpout);
 open(LOG, ">>/var/local/cgi-logs/mycgi-log")
 or die "Unable to append to mycgi-log: $!\n";
 carpout(*LOG);
}
```

Vous pouvez vous arranger pour que les erreurs fatales soient retournées au navigateur client, ceci vous permettant d'obtenir un meilleur débogage, mais pouvant paraître confus pour l'utilisateur final.

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Si l'erreur se produit avant même que vous ayez produit les en-têtes HTTP en sortie, le module essaiera de les générer pour éviter une erreur 500 du serveur. Les avertissements normaux continueront à être envoyés vers le fichier de log des erreurs du serveur (ou là où vous les avez envoyées via `carpout`) préfixés par le nom du script et la date.

### 26.1.4 Comment enlever les balises HTML d'une chaîne ?

La meilleure façon (mais pas obligatoirement la plus rapide) est d'utiliser `HTML::Parser` disponible sur le CPAN. Un autre moyen généralement correct est l'utilisation de `HTML::FormatText` qui non seulement retire le HTML mais aussi essaye de réaliser un petit formatage simple du texte brut résultant.

Plusieurs personnes essayent une approche simpliste utilisant des expressions rationnelles, comme `s/<.*?>//g`, mais ceci ne fonctionne pas correctement dans de nombreux cas car les marqueurs HTML peuvent continuer après des sauts de lignes, ils peuvent contenir des `<` ou des `>` entre guillemets, ou des commentaires HTML peuvent être présents. De plus, ces personnes oublient de convertir les entités comme `&lt;` ; par exemple.

Voici une "solution-basique", qui fonctionne avec la plupart des fichiers :

```
#!/usr/bin/perl -p0777
s/<(?:[^\>'"]*|(["']).*?\1)*>//gs
```

Si vous souhaitez une solution plus complète, regardez le programme `striphtml` se décomposant en 3 étapes sur [http://www.cpan.org/authors/Tom\\_Christiansen/scripts/striphtml.gz](http://www.cpan.org/authors/Tom_Christiansen/scripts/striphtml.gz).

Voici quelques pièges auxquels vous devriez penser avant de choisir une solution :

```
 B">
```

```
<IMG SRC = "foo.gif"
 ALT = "A > B">

<!-- <Un commentaire> -->

<script>if (a<b && a>c)</script>

<# Just data #>

<![INCLUDE CDATA [>>>>>>>>>>>>]]>
```

Si les commentaires HTML incluent d'autres balises, ces solutions échoueraient aussi sur un texte tel que celui-ci :

```
<!-- Cette section est en commentaire.
 You can't see me!
-->
```

### 26.1.5 Comment extraire des URL ?

Vous pouvez facilement extraire toutes sortes d'URL d'un document HTML en utilisant `HTML::SimpleLinkExtor` qui gère les ancres, les images, les objets, les cadres et plein d'autres balises qui peuvent contenir des URL. Si vous avez besoin de quelque chose de plus complexe, vous pouvez créer votre propre sous-classe en héritant de `HTML::LinkExtor` ou de `HTML::Parser`. Vous pouvez même utiliser `HTML::SimpleLinkExtor` comme base de travail pour répondre à vos besoins spécifiques.

Vous pouvez utiliser `URI::Find` pour extraire des URL d'un document texte quelconque.

Des solutions moins complètes basées sur des expressions rationnelles peuvent économiser du temps de calcul si vous êtes sûr que l'entrée est simple. Voici une solution, proposée par Tom Christiansen, qui va 100 fois plus vite que la plupart des solutions fournies par les modules mais qui n'extraient des URL que des ancres (les balises A) dont le seul attribut est un HREF :

```
#!/usr/bin/perl -n00
qxurl - tchrist@perl.com
print "$2\n" while m{
 < \s*
 A \s+ HREF \s* = \s* (["']) (.*) \1
 \s* >
}gsix;
```

### 26.1.6 Comment télécharger un fichier depuis la machine d'un utilisateur ? Comment ouvrir un fichier d'une autre machine ?

Dans ce cas, télécharger signifie utiliser la fonctionnalité d'envoi de fichier via un formulaire HTML. Vous autorisez ainsi l'utilisateur à choisir un fichier qui sera envoyé à votre serveur web. Pour vous, c'est un téléchargement alors que, pour l'utilisateur, c'est un envoi. Peu importe son nom, vous pourrez le faire en utilisant ce qu'on appelle l'encodage **multipart/form-data**. Le module `CGI.pm` (qui est un module standard de la distribution Perl) propose cette fonctionnalité dans via la méthode `start_multipart_form()` qui n'est pas la même que la méthode `startform()`.

Pour des exemples de codes et des informations supplémentaires, voir la section parlant du téléchargement de fichiers dans la documentation du module `CGI.pm`.

### 26.1.7 Comment faire un menu pop-up en HTML ?

Utiliser les tags **SELECT** et **OPTION**. Le module `CGI.pm` (disponible au CPAN) support cette fonctionnalité, ainsi que de nombreuses autres, incluant quelques-unes qui se synthétisent intelligemment d'elles-mêmes.



### 26.1.8 Comment récupérer un fichier HTML ?

Utiliser le module `LWP::Simple` disponible au CPAN, qui fait partie de l'excellent package `libwww-perl` (LWP). D'un autre côté, si vous avez le navigateur en mode texte `lynx` installé sur votre système, il n'est pas mauvais de faire :

```
$html_code = 'lynx -source $url';
$text_data = 'lynx -dump $url';
```

Les modules `libwww-perl` (LWP) du CPAN fournissent une façon plus puissante de le faire. Ils n'ont pas besoin de `lynx`, mais tout comme `lynx`, ils peuvent fonctionner à travers les serveurs mandataires (proxy, NDT) :

```
version la plus simple
use LWP::Simple;
$content = get($URL);

ou affiche du HTML depuis un URL
use LWP::Simple;
getprint "http://www.linpro.no/lwp/";

ou affiche de l'ASCII depuis le HTML d'un URL
nécessite aussi le paquetage HTML-Tree du CPAN
use LWP::Simple;
use HTML::Parser;
use HTML::FormatText;
my ($html, $ascii);
$html = get("http://www.perl.com/");
defined $html
 or die "Can't fetch HTML from http://www.perl.com/";
$ascii = HTML::FormatText->new->format(parse_html($html));
print $ascii;
```

### 26.1.9 Comment automatiser la soumission d'un formulaire HTML ?

Si vous souhaitez faire quelque chose complexe comme parcourir plusieurs pages et formulaires d'un site web, vous pouvez utiliser le module `WWW::Mechanize`. Voir sa documentation pour plus de détails.

Si vous soumettez des valeurs en utilisant la méthode GET, créez un URL et codez le formulaire en utilisant la méthode `query_form` :

```
use LWP::Simple;
use URI::URL;

my $url = url('http://www.perl.com/cgi-bin/cpan_mod');
$url->query_form(module => 'DB_File', readme => 1);
$content = get($url);
```

Si vous utilisez la méthode POST, créez votre propre agent utilisateur et codez le contenu de façon appropriée.

```
use HTTP::Request::Common qw(POST);
use LWP::UserAgent;

$ua = LWP::UserAgent->new();
my $req = POST 'http://www.perl.com/cgi-bin/cpan_mod',
 [module => 'DB_File', readme => 1];
$content = $ua->request($req)->as_string;
```

### 26.1.10 Comment décoder ou créer ces %-encodings sur le web ?

Si vous écrivez un script CGI, vous devriez utiliser le module CGI.pm qui vient avec perl ou un autre module équivalent. Le module CGI décode automatiquement les requêtes et propose la fonction encode() pour gérer l'encodage.

La meilleure source d'informations au sujet de l'encodage des URI est la RFC 2396. En principe, les substitutions suivantes le font bien :

```
s/([\w() '*~!.-])/sprintf '%x', ord $1/eg; # encodage
s/%([A-Fa-f\d]{2})/chr hex $1/eg; # décodage
s/%([:xdigit:]{2})/chr hex $1/eg; # idem
```

En revanche, vous ne devriez les appliquer qu'à des composants individuels d'un URI et non à l'URI en entier. Sinon vous risquez de perdre de l'information et donc de rater quelque chose. Si vous ne comprenez pas pourquoi, pas de panique. Consultez la section 2 de la RFC qui donne probablement les meilleures explications à ce sujet.

La RFC 2396 contient aussi plein d'autres informations intéressantes et en particulier des expressions rationnelles permettant de découper en composant un URI quelconque (annexe B).

### 26.1.11 Comment rediriger le navigateur vers une autre page ?

Spécifiez l'URL complet de la page de destination (même si elle est sur le même serveur). C'est l'une des deux sortes de réponses "Location:" prévues par les spécifications CGI pour un script "Parser Header". L'autre sorte qui retourne "Location:" avec un URL sous la forme d'un chemin absolu (sans le schéma ni le nom du serveur) est résolu en interne par le serveur sans utiliser la redirection HTTP. En tous cas, les spécifications CGI n'autorisent pas les URL relatifs.

L'utilisation de CGI.pm est fortement recommandée. L'exemple ci-dessous montre une redirection avec un URL complet. Cette redirection est gérée par le navigateur web.

```
use CGI qw/:standard/;

my $url = 'http://www.cpan.org/';
print redirect($url);
```

Ce deuxième exemple montre une redirect avec un URL sous la forme d'un chemin absolu. Cette redirect est gérée par le serveur web local.

```
my $url = '/CPAN/index.html';
print redirect($url);
```

Si vous souhaitez le coder vous-même directement, c'est faisable comme suit (le "\n" final est montré séparément pour être plus clair). que l'URL soit complet ou sous la forme d'un simple chemin absolu.

```
print "Location: $url\n"; # en-tête de la réponse CGI
print "\n"; # fin de l'en-tête
```

### 26.1.12 Comment mettre un mot de passe sur mes pages Web ?

Pour que l'authentification soit active sur votre serveur web, il faut le configurer pour cela. Cette configuration diffère selon le serveur web – apache ne fait pas comme iPlanet qui ne fait pas comme IIS. Consultez la documentation de votre serveur web pour connaître les détails de configuration de ce serveur.

### 26.1.13 Comment éditer mes fichiers .htpasswd et .htgroup en Perl ?

Les modules HTTPD::UserAdmin et HTTPD::GroupAdmin proposent une interface orientée objet cohérente pour ces fichiers, quelle que soit la façon dont ils sont stockés. Les bases de données peuvent être du texte, une dbm, une Berkley DB ou n'importe quelle autre base de données avec un pilote compatible DBI. HTTPD::UserAdmin accepte les fichiers utilisés par les mécanismes d'authentification 'Basic' et 'Digest'. Voici un exemple :

```
use HTTPD::UserAdmin ();
HTTPD::UserAdmin
->new(DB => "/foo/.htpasswd")
->add($username => $password);
```

### 26.1.14 Comment être sûr que les utilisateurs ne peuvent pas entrer de valeurs dans un formulaire qui font faire de vilaines choses à mon script CGI ?

Voyez les références à la sécurité dans la Meta FAQ CGI :

[http://www.perl.org/CGI\\_MetaFAQ.html](http://www.perl.org/CGI_MetaFAQ.html)

### 26.1.15 Comment analyser un en-tête de mail ?

Pour une solution rapide et peu propre, essayez cette solution dérivée de *split* in *perlfunc* :

```
$/ = '';
$header = <MSG>;
$header =~ s/\n\s+/ /g; # fusionne les lignes fractionnées
%head = (UNIX_FROM_LINE, split /^([\w-]+):\s*/m, $header);
```

Cette solution ne fonctionne pas correctement si, par exemple, vous essayez de mettre à jour toutes les lignes reçues. Une approche plus complète consiste à utiliser le module `Mail::Header` du CPAN (faisant parti du paquetage `MailTools`).

### 26.1.16 Comment décoder un formulaire CGI ?

(contribution de brian d foy)

Utilisez la module `CGI.pm` qui vient avec Perl. Il est rapide, simple et effectue le travail nécessaire pour s'assurer que tout se passe correctement. Il gère les requêtes GET, POST et HEAD, les formulaires multipart, les champs multi-valués, les combinaisons chaîne de requête et données de flux, et plein d'autres choses dont vous ne voulez probablement pas vous préoccuper.

Et tout cela d'une manière on ne peut plus simple : le module `CGI` analyse automatiquement les entrées et rend accessible chacune des valeurs via la fonction `param()`.

```
use CGI qw(:standard);

my $total = param('prix') + param('transport');
my @items = param('item'); # valeurs multiples, pour un même champ
```

Si vous préférez une interface orientée objet, `CGI.pm` sait aussi le faire.

```
use CGI;

my $cgi = CGI->new();

my $total = $cgi->param('prix') + $cgi->param('transport');

my @items = $cgi->param('item');
```

Vous pouvez aussi essayer le module `CGI::Minimal` qui est version allégée pour faire la même chose. D'autres modules `CGI::*` de CPAN peuvent aussi mieux vous convenir.

Beaucoup de gens essaye d'écrire leur propre décodeur (ou recopie celui d'un autre programme) et tombe ensuite dans l'un des nombreux pièges de cette tâche. Il est plus simple et moins embêtant d'utiliser `CGI.pm`.

### 26.1.17 Comment vérifier la validité d'une adresse électronique ?

Vous ne pouvez pas, du moins pas en temps réel. Dommage, hein ?

Sans envoyer un mail à cette adresse pour constater qu'il y a un être humain à l'autre bout pour vous répondre, vous ne pouvez pas déterminer si une adresse est valide. Même si vous appliquez l'en-tête standard d'email, vous pouvez rencontrer des problèmes, car il existe des adresses valides qui ne sont pas compatibles avec la RFC-822 (le standard des en-têtes des mails), et inversement il existe des adresses qui ne sont pas délivrables et qui sont compatibles.

Vous pouvez utiliser les modules `Email::Valid` ou `RFC::RFC822::Address` qui vérifient le format de l'adresse bien que cela ne garantit en rien que l'adresse est valide (c'est à dire qu'un message envoyé ne sera pas refusé). Les modules comme `Mail::CheckUser` et `Mail::EXPN` interrogent le système de noms de domaines et les serveurs de messagerie pour tenter d'en apprendre plus mais cela ne marche pas partout – surtout si l'administrateur du domaine gère consciencieusement la sécurité de son site.

Beaucoup sont tentés d'éliminer les adresses email invalides en se basant sur une expression régulière simple, comme `/^[\w.-]+\@(?:[\w-]+\.)+\w+$/`. C'est une très mauvaise idée. Car on rejette ainsi beaucoup d'adresses valides et que cela ne garantit en rien que les adresses acceptées sont effectivement délivrables, ce n'est donc pas conseillé. À la place, regardez plutôt [http://www.cpan.org/authors/Tom\\_Christiansen/scripts/ckaddr.gz](http://www.cpan.org/authors/Tom_Christiansen/scripts/ckaddr.gz) qui effectivement vérifie une compatibilité complète avec les spécifications RFC (excepté les commentaires imbriqués), vérifie qu'il ne s'agit pas d'une adresse que vous ne désirez pas (cad, Bill Clinton ou votre responsable de compte mail), puis s'assure que le nom d'hôte donné peut être trouvé dans les enregistrements MX du DNS. Ce n'est pas très rapide, mais ça marche pour ce que ça essaye de faire.

Notre meilleur conseil pour vérifier l'adresse de quelqu'un est de lui faire entrer deux fois son adresse, tout comme vous le faites pour changer un mot de passe. Ceci élimine habituellement les fautes de frappe. Si les deux versions sont égales, envoyez un courrier à cette adresse avec un message personnel ayant cette allure :

```
Dear someuser@host.com,
```

```
Please confirm the mail address you gave us Wed May 6 09:38:41
MDT 1998 by replying to this message. Include the string
"Rumpelstiltskin" in that reply, but spelled in reverse; that is,
start with "Nik...". Once this is done, your confirmed address will
be entered into our records.
```

Si vous recevez le message et s'ils ont suivi vos indications, vous pouvez être raisonnablement assuré que l'adresse est réelle.

Une stratégie proche moins ouverte à la tromperie est de leur donner un PIN (numéro d'identification personnel). Enregistrez l'adresse et le PIN (le mieux est qu'il soit aléatoire) pour un traitement ultérieur. Dans le message que vous envoyez, demandez-leur d'inclure le PIN dans leur réponse. Mais si le message rebondit, ou est inclus automatiquement par un script "vacation" (en vacances), il sera là de toute façon. Il est donc plus efficace de leur demander de renvoyer un PIN légèrement modifié, par exemple inversé, ou une unité ajoutée à chaque chiffre, etc.

### 26.1.18 Comment décoder une chaîne MIME/BASE64 ?

Le module `MIME::Base64` (disponible sur CPAN) gère cela ainsi que l'encodage MIME/QP. Décoder du BASE64 devient alors aussi simple que :

```
use MIME::Base64;
$decoded = decode_base64($encoded);
```

Le paquetage `MIME-tools` (disponible sur CPAN) propose ces extractions avec en sus le décodage des documents attachés encodés en BASE64, tout cela directement depuis un message e-mail.

Si la chaîne à décoder est courte (moins de 84 caractères), une approche plus directe consiste à utiliser la fonction `unpack()` avec le formatage "u" après quelques translations mineures :

```
tr#A-Za-z0-9+/#cd; # supprime les caractères non base-64
tr#A-Za-z0-9+/# -_#; # convertit dans le format uuencode
$len = pack("c", 32 + 0.75*length); # calcule la longueur en octets
print unpack("u", $len . $_); # uudécode et affiche
```

### 26.1.19 Comment renvoyer l'adresse électronique de l'utilisateur ?

Sur les systèmes supportant `getpwuid`, et donc la variable `$<`, ainsi que le module `Sys::Hostname` (qui fait partie de la distribution standard de Perl), vous pouvez probablement essayer d'utiliser quelque chose comme ceci :

```
use Sys::Hostname;
$address = sprintf('%s@%s', getpwuid($<), hostname);
```

La politique de la compagnie sur les adresses email peut signifier que ceci génère des adresses que le système de mail de la compagnie n'acceptera pas, ainsi vous devriez demander les adresses email des utilisateurs quand ceci compte. Qui plus est, tous les systèmes sur lesquels fonctionne Perl n'acceptent pas ces informations comme sur Unix.

Le module `Mail::Util` du CPAN (faisant partie du package `MailTools`) procure une fonction `mailaddress()` qui essaye de créer l'adresse email d'un utilisateur. Il effectue une démarche plus intelligente que le code précédent, utilisant des informations fournies quand le module a été installé, mais cela peut rester incorrect. Encore une fois, la meilleure manière est souvent de simplement poser la question à l'utilisateur.

### 26.1.20 Comment envoyer un mail ?

Utilisez directement le programme `sendmail` :

```
open(SENDMAIL, "|/usr/lib/sendmail -oi -t -odq")
 or die "Can't fork for sendmail: $!\n";
print SENDMAIL <<"EOF";
From: User Originating Mail <me\@host>
To: Final Destination <you\@otherhost>
Subject: A relevant subject line

Body of the message goes here after the blank line
in as many lines as you like.
EOF
close(SENDMAIL) or warn "sendmail didn't close nicely";
```

L'option `-oi` empêche `sendmail` d'interpréter une ligne constituée d'un seul point comme une "fin de message". L'option `-t` lui dit d'utiliser les en-têtes pour décider à qui envoyer le message, et `-odq` lui dit de placer le message dans la file d'attente. Cette dernière option signifie que votre message ne sera pas immédiatement envoyé, donc ne la mettez pas si vous voulez un envoi immédiat.

Alternativement, des approches moins pratiques comprennent l'appel direct à mail (parfois appelé `mailx`) ou la simple ouverture du port 25 pour avoir une conversation intime rien qu'entre vous et le démon SMTP distant, probablement `sendmail`.

Ou vous pourriez utiliser le module `Mail::Mailer` du CPAN :

```
use Mail::Mailer;

$mailer = Mail::Mailer->new();
$mailer->open({ From => $from_address,
 To => $to_address,
 Subject => $subject,
 })
 or die "Can't open: $!\n";
print $mailer $body;
$mailer->close();
```

Le module `Mail::Internet` utilise `Net::SMTP` qui est moins Unix-centrique que `Mail::Mailer`, mais moins fiable. Évitez les commandes SMTP crues. Il y a de nombreuses raisons pour utiliser un agent de transport de mail comme `sendmail`. Celles-ci comprennent la mise en file d'attente, les enregistrements MX et la sécurité.

### 26.1.21 Comment utiliser MIME pour attacher des documents à un mail ?

Cette réponse est directement extraite de la documentation du module MIME::Lite. Créez un message multipart (c'est à dire avec des documents attachés).

```
use MIME::Lite;

Création d'un nouveau message multipart
$msg = MIME::Lite->new(
 From =>'me@myhost.com',
 To =>'you@yourhost.com',
 Cc =>'some@other.com, some@more.com',
 Subject =>'A message with 2 parts...',
 Type =>'multipart/mixed'
);

Ajout de parties (chaque "attach" a les mêmes arguments que "new"):
$msg->attach(Type =>'TEXT',
 Data =>"Here's the GIF file you wanted"
);
$msg->attach(Type =>'image/gif',
 Path =>'aaa000123.gif',
 Filename =>'logo.gif'
);

$text = $msg->as_string;
```

MIME::Lite propose aussi une méthode pour envoyer le messages.

```
$msg->send;
```

Par défaut, il utilise sendmail mais on peut le paramétrer pour qu'il utilise SMTP via *Net::SMTP*.

### 26.1.22 Comment lire du courrier ?

Vous pourriez utiliser le module Mail::Folder de CPAN (faisant partie du paquetage MailFolder) ou le module Mail::Internet de CPAN (faisant partie du paquetage MailTools), toutefois, un module est souvent un marteau pour écraser une mouche. Voici un trieur de mail.

```
#!/usr/bin/perl

my(@msgs, @sub);
my $msgno = -1;
$/ = '';
lit un paragraphe
while (<>) {
 if (/^From/m) {
 /^Subject:\s*(?:Re:\s*)*(.*)/mi;
 $sub[++$msgno] = lc($1) || '';
 }
 $msgs[$msgno] .= $_;
}
for my $i (sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msgs)) {
 print $msgs[$i];
}
```

Ou de façon plus succincte,

```
#!/usr/bin/perl -n00
bysub2 - awkish sort-by-subject
BEGIN { $msgno = -1 }
$sub[++$msgno] = (/^Subject:\s*(?:Re:\s*)*(.*)/mi)[0] if /^From/m;
$msg[$msgno] .= $_;
END { print @msg[sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msg)] }
```

### 26.1.23 Comment trouver mon nom de machine / nom de domaine / mon adresse IP ?

(contribution de brian d foy)

Le module `Net::Domain`, qui fait partie de la distribution standard depuis Perl 5.7.3, peut vous fournir le nom d'hôte complet (fully qualified domain name ou FQDN), le nom d'hôte (ou de machine) ou le nom de domaine.

```
use Net::Domain qw(hostname hostfqdn hostdomain);

my $host = hostfqdn();
```

Le module `Sys::Hostname`, inclus dans la distribution standard depuis Perl 5.6, peut aussi vous donner le nom d'hôte.

```
use Sys::Hostname;

$host = hostname();
```

Pour obtenir l'adresse IP, vous pouvez faire appel à la fonction prédéfinie `<gethostbyname>` pour transformer le nom d'hôte en un nombre. Pour convertir ce nombre en une adresse classique (a.b.c.d) que la plupart des gens attendent, utilisez la fonction `inet_ntoa` du module `Socket` qui fait partie de perl.

```
use Socket;

my $address = inet_ntoa(
 scalar gethostbyname($host || 'localhost')
);
```

### 26.1.24 Comment récupérer un article de news ou les groupes actifs ?

Utilisez les modules `Net::NNTP` ou `News::NNTPClient`, tous les deux disponibles sur le CPAN. Ceux-ci peuvent rendre des tâches comme la récupération de la liste des groupes de news aussi simple que :

```
perl -MNews::NNTPClient
 -e 'print News::NNTPClient->new->list("newsgroups")'
```

### 26.1.25 Comment récupérer/envoyer un fichier par FTP ?

`LWP::Simple` (disponible sur le CPAN) peut récupérer mais pas envoyer. `Net::FTP` (aussi disponible sur le CPAN) est plus complexe, mais peut envoyer aussi bien que récupérer.

### 26.1.26 Comment faire du RPC en Perl ?

(contribution de brian d foy)

Utilisez l'un des modules RPC disponibles sur CPAN (<http://search.cpan.org/search?query=RPC&mode=all>).

## 26.2 AUTEUR ET COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen, Nathan Torkington et autres auteurs cités. Tous droits réservés.

Cette documentation est libre. Vous pouvez la redistribuer et/ou la modifier sous les mêmes conditions que Perl lui-même. Indépendamment de sa distribution, tous les exemples de code de ce fichier sont ici placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre plaisir ou pour un profit. Un simple commentaire dans le code en précisant l'origine serait de bonne courtoisie mais n'est pas obligatoire.

## 26.3 TRADUCTION

### 26.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

**26.3.2 Traducteur**

Traduction initiale : Aymeric Barantal <Aymeric.Barantal@grolier.fr>. Mise à jour : Paul Gaborit <paul.gaborit@enstimac.fr>.

**26.3.3 Relecture**

Régis Julié <Regis.Julie@cetelem.fr>, Roland Trique <roland.trique@uhb.fr> (mise à jour), Gérard Delafond.



**Troisième partie**

**Manuel de référence**

# Chapitre 27

## perlsyn

Syntaxe de Perl

### 27.1 DESCRIPTION

Un script Perl est constitué d'une suite de déclarations et d'instructions qui sont exécutées de haut en bas. Les boucles, les sous-programmes et d'autres structures de contrôles vous permettent de vous déplacer dans le code.

Perl est un langage à syntaxe libre. Vous pouvez donc le présenter et l'indenter comme bon vous semble. Les espaces ne servent qu'à séparer les éléments syntaxiques contrairement à des langages comme Python où ils font partie intégrante de la syntaxe.

La plupart des éléments syntaxiques de Perl sont **optionnels**. Au lieu de vous obliger à mettre des parenthèses pour chaque appel de fonction ou à déclarer toutes les variables, Perl vous laisse libre de le faire ou non et se débrouille pour comprendre ce que vous voulez. Ceci s'appelle **Fait Ce Que Je Pense** ou **FCQJP** (NdT: en anglais **Do What I Mean** ou **DWIM**). Le programmeur peut donc être  **paresseux**  et peut coder dans le style qui lui plait.

Perl **emprunte sa syntaxe** et ses concepts à de nombreux langages : awk, sed, C, Bourne Shell, Smalltalk, Lisp et même l'Anglais. D'autres langages ont emprunté à la syntaxe de Perl, en particulier ses extensions aux expressions rationnelles. Donc, si vous avez programmé avec d'autres langages, vous rencontrerez des constructions familières en Perl. Souvent elles fonctionnent de la même manière mais lisez tout de même *perltrap* pour savoir quand elles diffèrent.

#### 27.1.1 Déclarations

Les seules choses que vous devez absolument déclarer en Perl sont les formats de rapport et les sous-programmes (parfois ce n'est même pas nécessaire pour les sous-programmes). Une variable contient la valeur indéfinie (`undef`) jusqu'à ce qu'on lui affecte une valeur, qui est n'importe quoi autre que `undef`. Lorsqu'il est utilisé comme un nombre, `undef` est traité comme `0`; lorsqu'il est utilisé comme une chaîne, il est traité comme la chaîne vide, `""`; et lorsqu'il est utilisé comme une référence qui n'a pas été affectée, il est traité comme une erreur. Si vous activez les avertissements, vous serez notifié de l'utilisation d'une valeur non initialisée chaque fois que vous traiterez `undef` comme une chaîne ou un nombre. En tout cas, habituellement. Son utilisation dans un contexte booléen, tel que :

```
my $a;
if ($a) {}
```

ne déclenche pas d'avertissement (parce qu'on teste la véracité et non la présence d'une valeur définie). Les opérateurs tels que `++`, `-`, `+=`, `-=` et `.=` lorsqu'ils agissent sur une valeur indéfinies comme dans :

```
my $a;
$a++;
```

ne déclenchent pas non plus de tels avertissements.

Une déclaration peut être mise partout où une instruction peut trouver place, mais n'a pas d'effet sur l'exécution de la séquence d'instructions principale - les déclarations prennent toutes effet au moment de la compilation. Typiquement, toutes les déclarations sont placées au début ou à la fin du script. Toutefois, si vous utilisez des variables privées de portée

lexicales créées avec `my()`, vous devez vous assurer que la définition de votre format ou de votre sous-programme est à l'intérieur du même bloc que le `my` si vous voulez pouvoir accéder à ces variables privées.

La déclaration d'un sous-programme permet à un nom de sous-programme d'être utilisé comme s'il était un opérateur de liste à partir de ce point dans le programme. Vous pouvez déclarer un sous-programme sans le définir en disant `sub name`, ainsi :

```
sub myname;
$me = myname $0 or die "can't get myname";
```

Notez que `myname()` fonctionne comme un opérateur de liste, et non comme un opérateur unaire ; faites donc attention à utiliser `or` au lieu de `||` dans ce cas. Toutefois, si vous déclarez le sous-programme avec `sub myname ($)`, alors `myname` fonctionnerait comme un opérateur unaire, donc `or` aussi bien que `||` feraient l'affaire.

Les déclarations de sous-programmes peuvent aussi être chargées à l'aide de l'instruction `require` ou bien à la fois chargées et importées dans votre espace de noms via l'instruction `use`. Voir *perlmod* pour plus de détails.

Une séquence d'instructions peut contenir des déclarations de variables de portée lexicale, mais à part pour déclarer un nom de variable, la déclaration fonctionne comme une instruction ordinaire, et est élaborée à l'intérieur de la séquence d'instructions en tant que telle. Cela signifie qu'elle a à la fois un effet à la compilation et lors de l'exécution.

### 27.1.2 Commentaires

Dans une ligne, tout texte commençant par le caractère `"#"` et jusqu'à la fin de la ligne est considéré comme un commentaire et est donc ignoré. L'exception à cette règle concerne les `"#"` présents dans les chaînes de caractères ou dans les expressions rationnelles.

### 27.1.3 Instructions simples

Le seul type d'instruction simple est une expression évaluée pour ses effets de bord. Chaque instruction simple doit être terminée par un point-virgule, sauf si c'est la dernière instruction d'un bloc, auquel cas le point-virgule est optionnel. (Nous vous encourageons tout de même à placer ce point-virgule si le bloc prend plus d'une ligne, car vous pourriez éventuellement ajouter une autre ligne). Notez qu'il existe des opérateurs comme `eval {}` et `do {}` qui ont l'air d'instructions composées, mais qui ne le sont pas (ce sont juste des TERMES dans une expression), et qui ont donc besoin d'une terminaison explicite s'ils sont utilisés comme dernier élément d'une instruction.

### 27.1.4 Le vrai et le faux

Le nombre 0, les chaînes `'0'` et `"`, la liste vide `()` et `undef` sont considérés comme faux dans un contexte booléen. Tout autre valeur est considérée comme vraie. La négation d'une valeur vraie par les opérateurs `!` ou `not` retourne une valeur fautive spéciale. Lorsqu'elle est évaluée en tant que chaîne, elle vaut `"` mais évaluée en tant que nombre, elle vaut 0.

### 27.1.5 Modificateurs d'instruction

Toute instruction simple peut être suivie de façon optionnelle par un *UNIQUE* modificateur, juste avant le point-virgule de terminaison (ou la fin du bloc). Les modificateurs possibles sont :

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LISTE
```

L'expression `EXPR` qui suit le modificateur s'appelle la "condition". Sa véracité ou sa fausseté détermine le comportement du modificateur.

`if` exécute l'instruction une fois *si et seulement si* la condition est vraie. `unless` fait le contraire : il exécute l'instruction *sauf si* la condition est vraie (autrement dit, si la condition est fautive).

```
print "Les bassets ont de longues oreilles" if length $oreilles >= 10;
aller_dehors() and jouer() unless $il_pleut;
```

Le modificateur `foreach` est un itérateur : il exécute l'instruction une fois pour chacun des items de LISTE (avec `$_` qui est un alias de l'item courant).

```
print "Bonjour $_ !\n" foreach ("tout le monde", "Anne", "Maitresse");
```

`while` répète l'instruction *tant que* la condition est vraie. `until` fait le contraire : il répète l'instruction *jusqu'à* ce que la condition soit vraie (ou, autrement dit, tant que la condition est fausse).

```
Dans les deux cas on compte de 0 à 10
print $i++ while $i <= 10;
print $j++ until $j > 10;
```

Les modificateurs `while` et `until` ont la sémantique habituelle des "boucles `while`" (la condition est évaluée en premier), sauf lorsqu'ils sont appliqués à un `do-BLOC` (ou à la construction désapprouvée `do-SOUS_PROGRAMME`), auquel cas le bloc s'exécute une fois avant que la condition ne soit évaluée. Ceci afin que vous puissiez écrire des boucles telles que :

```
do {
 $line = <STDIN>;
 ...
} until $line eq ".\n";
```

Voir `do` in *perlfunc*. Notez aussi que l'instruction de contrôle de boucle décrite plus tard *ne* fonctionnera pas dans cette construction, car les modificateurs n'utilisent pas de labels de boucle. Désolé. Vous pouvez toujours mettre un autre bloc à l'intérieur (for `next`) ou autour (for `last`) pour réaliser ce genre de choses. Pour `next`, il suffit de doubler les accolades :

```
do {{
 next if $x == $y;
 # faire quelque chose ici
}} until $x++ > $z;
```

Pour `last`, vous devez faire quelque chose de plus élaboré :

```
LOOP: {
 do {
 last if $x = $y**2;
 # faire quelque chose ici
 } while $x++ <= $z;
}
```

**NOTE :** le comportement d'une instruction `my` modifié par un modificateur conditionnel ou une construction de boucle (par exemple `my $x if ...`) est **indéfini**. La valeur de la variable peut être `undef`, la valeur précédemment affectée ou n'importe quoi d'autre. Ne dépendez pas d'un comportement particulier. Les prochaines versions de perl feront peut-être quelque chose de différents que celle que vous utilisez actuellement.

### 27.1.6 Instructions composées

En Perl, une séquence d'instructions qui définit une portée est appelée un bloc. Un bloc est parfois délimité par le fichier qui le contient (dans le cas d'un fichier requis, ou dans celui du programme en entier), et parfois un bloc est délimité par la longueur d'une chaîne (dans le cas d'un `eval`).

Mais généralement, un bloc est délimité par des accolades. Nous appellerons cette construction syntaxique un BLOC.

Les instructions composées suivantes peuvent être utilisées pour contrôler un flux :

```
if (EXPR) BLOC
if (EXPR) BLOC else BLOC
if (EXPR) BLOC elsif (EXPR) BLOC ... else BLOC
LABEL while (EXPR) BLOC
LABEL while (EXPR) BLOC continue BLOC
LABEL until (EXPR) BLOC
LABEL until (EXPR) BLOC continue BLOC
LABEL for (EXPR; EXPR; EXPR) BLOC
LABEL foreach VAR (LIST) BLOC
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOC continue BLOC
```

Notez que, contrairement au C et au Pascal, tout ceci est défini en termes de BLOCs, et non d'instructions. Ceci veut dire que les accolades sont *requis* - aucune instruction ne doit traîner. Si vous désirez écrire des conditionnelles sans accolades, il existe plusieurs autres façons de le faire. Les exemples suivants font tous la même chose :

```
if (!open(F00)) { die "Can't open $F00: $!"; }
die "Can't open $F00: $!" unless open(F00);
open(F00) or die "Can't open $F00: $!"; # F00 or bust!
open(F00) ? 'hi mom' : die "Can't open $F00: $!";
 # ce dernier est un peu exotique
```

L'instruction `if` est directe. Puisque les BLOCs sont toujours entourés d'accolades, il n'y a jamais d'ambiguïté pour savoir à quel `if` correspond un `else`. Si vous utilisez `unless` à la place de `if`, le sens du test est inversé.

L'instruction `while` exécute le bloc tant que l'expression est vraie (son évaluation ne renvoie pas une chaîne nulle ("") ou `0` ou `"0"`). L'instruction `until` exécute le bloc tant que l'expression est fautive. Le LABEL est optionnel, et s'il est présent, il est constitué d'un identifiant suivi de deux points. Le LABEL identifie la boucle pour les instructions de contrôle de boucle `next`, `last`, et `redo`. Si le LABEL est omis, l'instruction de contrôle de boucle se réfère à la boucle incluse dans toutes les autres. Ceci peut amener une recherche dynamique dans votre pile au moment de l'exécution pour trouver le LABEL. Un comportement aussi désespérant provoquera un avertissement si vous utilisez le pragma `use warnings` ou l'option `-w`.

S'il existe un BLOC `continue`, il est toujours exécuté juste avant que la condition ne soit à nouveau évaluée. Ce bloc peut donc être utilisé pour incrémenter une variable de boucle, même lorsque la boucle a été continuée via l'instruction `next`.

### 27.1.7 Contrôle de boucle

La commande `next` démarre la prochaine itération de la boucle :

```
LINE: while (<STDIN>) {
 next LINE if /^#/; # elimine les commentaires
 ...
}
```

La commande `last` sort immédiatement de la boucle en question. Le bloc `continue`, s'il existe, n'est pas exécuté :

```
LINE: while (<STDIN>) {
 last LINE if /^$/; # sort quand on en a fini avec l'en-tete
 ...
}
```

La commande `redo` redémarre le bloc de la boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est *pas* exécuté. Cette commande est normalement utilisée par les programmes qui veulent se mentir à eux-mêmes au sujet de ce qui vient de leur être fourni en entrée.

Par exemple, lors du traitement d'un fichier comme */etc/termcap*. Si vos lignes en entrée sont susceptibles de se terminer par un antislash pour indiquer leur continuation, vous pouvez vouloir poursuivre et récupérer l'enregistrement suivant.

```
while (<>) {
 chomp;
 if (s/\\$/ /) {
 $_ .= <>;
 redo unless eof();
 }
 # on traite $_
}
```

qui est le raccourci Perl pour la version plus explicite :

```

LINE: while (defined($line = <ARGV>)) {
 chomp($line);
 if ($line =~ s/\\$/ /) {
 $line .= <ARGV>;
 redo LINE unless eof(); # pas eof(ARGV)!
 }
 # on traite $line
}

```

Notez que s'il y avait un bloc `continue` dans le code ci-dessus, il ne serait exécuté que pour les lignes rejetées par l'expression rationnelle (puisque `redo` saute le bloc `continue`). Un bloc `continue` est souvent utilisé pour réinitialiser les compteurs de lignes ou les recherches de motifs `?pat?` qui ne correspondent qu'une fois :

```

inspire par :1,$g/fred/s//WILMA/
while (<>) {
 ?(fred)? && s//WILMA $1 WILMA/;
 ?(barney)? && s//BETTY $1 BETTY/;
 ?(homer)? && s//MARGE $1 MARGE/;
} continue {
 print "$ARGV $.: $_";
 close ARGV if eof(); # réinitialise $.
 reset if eof(); # réinitialise ?pat?
}

```

Si le mot `while` est remplacé par le mot `until`, le sens du test est inversé, mais la condition est toujours testée avant la première itération.

Les instructions de contrôle de boucle ne fonctionnent pas dans un `if` ou dans un `unless`, puisque ce ne sont pas des boucles. Vous pouvez toutefois doubler les accolades pour qu'elles le deviennent.

```

if (/pattern/) {{
 last if /fred/;
 next if /barney/; # même effet que "last", mais en moins compréhensible
 # mettre quelque chose ici
}}

```

Ceci est du au fait qu'un bloc est considéré comme une boucle qui ne s'exécute qu'une fois. Voir BLOCs de base et instruction `switch` (§27.1.10).

La forme `while/if BLOC BLOC`, disponible en Perl 4, ne l'est plus. Remplacez toutes les occurrences de `if BLOC` par `if (do BLOC)`.

### 27.1.8 Boucles for

Les boucles `for` de Perl dans le style de C fonctionnent de la même façon que les boucles `while` correspondantes ; cela signifie que ceci :

```

for ($i = 1; $i < 10; $i++) {
 ...
}

```

est la même chose que ça :

```

$i = 1;
while ($i < 10) {
 ...
} continue {
 $i++;
}

```

Il existe une différence mineure : si des variables sont déclarées par `my` dans la section d'initialisation d'un `for`, la portée lexicale de ces variables est limitée à la boucle `for` (le corps de la boucle et sa section de contrôle).

En plus du bouclage classique dans les indices d'un tableau, `for` peut se prêter à de nombreuses autres applications intéressantes. En voici une qui évite le problème que vous rencontrez si vous testez explicitement la fin d'un fichier sur un descripteur de fichier interactif, ce qui donne l'impression que votre programme se gèle.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for (prompt(); <STDIN>; prompt()) {
 # faire quelque chose ici
}
```

Utiliser `readline` (ou sous sa forme d'opérateur, `<EXPR>`) comme condition d'un `for` est un raccourci d'écriture pour ce qui suit. Ce comportement est le même pour la condition d'une boucle `while`.

```
for (prompt(); defined($_ = <STDIN>); prompt()) {
 # faire quelque chose
}
```

### 27.1.9 Boucles `foreach`

La boucle `foreach` itère sur une liste de valeurs normale et fixe la variable `VAR` à chacune de ces valeurs successivement. Si la variable est précédée du mot-clé `my`, alors elle a une portée limitée du point de vue lexical, et n'est par conséquent visible qu'à l'intérieur de la boucle. Autrement, la variable est implicitement locale à la boucle et reprend sa valeur précédente à la sortie de la boucle. Si la variable était précédemment déclaré par `my`, elle utilise cette variable au lieu de celle qui est globale, mais elle est toujours locale à la boucle.

Le mot-clé `foreach` est en fait un synonyme du mot-clé `for`, vous pouvez donc utiliser `foreach` pour sa lisibilité ou `for` pour sa concision (Ou parce que le Bourne shell vous est plus familier que `csh`, vous rendant l'utilisation de `for` plus naturelle). Si `VAR` est omis, `$_` est fixée à chaque valeur.

Si un élément de `LIST` est une lvalue (une valeur modifiable), vous pouvez la modifier en modifiant `VAR` à l'intérieur de la boucle. À l'inverse, si un élément de `LIST` n'est pas une lvalue (c'est une constante), toute tentative de modification de cet élément échouera. En d'autres termes, la variable d'index de la boucle `foreach` est un alias implicite de chaque élément de la liste sur laquelle vous bouclez.

Si une partie de `LIST` est un tableau, `foreach` sera très troublé dans le cas où vous lui ajouteriez ou retireriez des éléments à l'intérieur de la boucle, par exemple à l'aide de `splice`. Ne faites donc pas cela.

`foreach` ne fera probablement pas ce que vous désirez si `VAR` est une variable liée ou une autre variable spéciale. Ne faites pas cela non plus.

Exemples :

```
for (@ary) { s/foo/bar/ }

for my $elem (@elements) {
 $elem *= 2;
}

for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
 print $count, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\n:]*/, $ENV{TERMCAP})) {
 print "Item: $item\n";
}
```

Voici comment un programmeur C pourrait coder un algorithme en Perl :

```

for (my $i = 0; $i < @ary1; $i++) {
 for (my $j = 0; $j < @ary2; $j++) {
 if ($ary1[$i] > $ary2[$j]) {
 last; # ne peut pas sortir totalement :-C
 }
 $ary1[$i] += $ary2[$j];
 }
 # voici l'endroit ou ce last m'emmene
}

```

Tandis que voici comment un programmeur Perl plus à l'aise avec l'idiome pourrait le faire :

```

OUTER: for my $wid (@ary1) {
INNER: for my $jet (@ary2) {
 next OUTER if $wid > $jet;
 $wid += $jet;
 }
}

```

Vous voyez à quel point c'est plus facile ? C'est plus propre, plus sûr, et plus rapide. C'est plus propre parce qu'il y a moins de bruit. C'est plus sûr car si du code est ajouté entre les deux boucles par la suite, le nouveau code ne sera pas exécuté accidentellement. Le `next` itère de façon explicite sur l'autre boucle plutôt que de simplement terminer celle qui est à l'intérieur. Et c'est plus rapide parce que Perl exécute une instruction `foreach` plus rapidement qu'une boucle `for` équivalente.

### 27.1.10 BLOCs de base et instruction `switch`

Un BLOC en lui-même (avec ou sans label) est d'un point de vue sémantique, équivalent à une boucle qui s'exécute une fois. Vous pouvez donc y utiliser n'importe quelle instruction de contrôle de boucle pour en sortir ou le recommencer (Notez que ce n'est *PAS* vrai pour les blocs `eval{}`, `sub{}`, ou `do{}` contrairement à la croyance populaire, qui *NE* comptent *PAS* pour des boucles). Le bloc `continue` est optionnel.

La construction de BLOC est particulièrement élégante pour créer des structures de choix.

```

SWITCH: {
 if (/^abc/) { $abc = 1; last SWITCH; }
 if (/^def/) { $def = 1; last SWITCH; }
 if (/^xyz/) { $xyz = 1; last SWITCH; }
 $nothing = 1;
}

```

Il n'y a pas d'instruction `switch` officielle en Perl, car il existe déjà plusieurs façons d'écrire quelque chose d'équivalent. En revanche, depuis Perl 5.8 pour obtenir les instructions `switch` et `case`, ceux qui le souhaitent peuvent faire appel à l'extension `Switch` en écrivant :

```
use Switch;
```

À partir de là, les nouvelles instructions seront disponibles. Ce n'est pas aussi rapide que cela pourrait l'être puisque elles ne font pas réellement partie du langage (elles sont réalisées via un filtrage des sources) mais elles sont disponibles et utilisables.

Vous pourriez écrire à la place du bloc précédent :

```

SWITCH: {
 $abc = 1, last SWITCH if /^abc/;
 $def = 1, last SWITCH if /^def/;
 $xyz = 1, last SWITCH if /^xyz/;
 $nothing = 1;
}

```



(Ce n'est pas aussi étrange que cela en a l'air une fois que vous avez réalisé que vous pouvez utiliser des "opérateurs" de contrôle de boucle à l'intérieur d'une expression, c'est juste l'opérateur binaire virgule normal utilisé dans un contexte scalaire. Voir Opérateur virgule in *perlop*.)

ou

```
SWITCH: {
 /^abc/ && do { $abc = 1; last SWITCH; };
 /^def/ && do { $def = 1; last SWITCH; };
 /^xyz/ && do { $xyz = 1; last SWITCH; };
 $nothing = 1;
}
```

ou formaté de façon à avoir un peu plus l'air d'une instruction switch "convenable" :

```
SWITCH: {
 /^abc/ && do {
 $abc = 1;
 last SWITCH;
 };

 /^def/ && do {
 $def = 1;
 last SWITCH;
 };

 /^xyz/ && do {
 $xyz = 1;
 last SWITCH;
 };

 $nothing = 1;
}
```

ou

```
SWITCH: {
 /^abc/ and $abc = 1, last SWITCH;
 /^def/ and $def = 1, last SWITCH;
 /^xyz/ and $xyz = 1, last SWITCH;
 $nothing = 1;
}
```

or même, horreur,

```
if (/^abc/)
 { $abc = 1 }
elseif (/^def/)
 { $def = 1 }
elseif (/^xyz/)
 { $xyz = 1 }
else
 { $nothing = 1 }
```

Un idiome courant pour une instruction switch est d'utiliser l'aliasing de foreach pour effectuer une affectation temporaire de \$\_ pour une reconnaissance pratique des cas :

```
SWITCH: for ($where) {
 /In Card Names/ && do { push @flags, '-e'; last; };
 /Anywhere/ && do { push @flags, '-h'; last; };
 /In Rulings/ && do { last; };
 die "unknown value for form variable where: '$where'";
}
```

Une autre approche intéressante de l'instruction `switch` est de s'arranger pour qu'un bloc `do` renvoie la valeur correcte :

```
$amode = do {
 if ($flag & O_RDONLY) { "r" } # XXX : n'est-ce pas 0?
 elsif ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
 elsif ($flag & O_RDWR) {
 if ($flag & O_CREAT) { "w+" }
 else { ($flag & O_APPEND) ? "a+" : "r+" }
 }
};
```

ou

```
print do {
 ($flags & O_WRONLY) ? "write-only" :
 ($flags & O_RDWR) ? "read-write" :
 "read-only";
};
```

Ou si vous êtes certain que toutes les clauses `&&` sont vraies, vous pouvez utiliser quelque chose comme ceci, qui "switch" sur la valeur de la variable d'environnement `HTTP_USER_AGENT`.

```
#!/usr/bin/perl
choisir une page du jargon file selon le browser
$dir = 'http://www.wins.uva.nl/~mes/jargon';
for ($ENV{HTTP_USER_AGENT}) {
 $page = /Mac/ && 'm/Macintrash.html'
 || /Win(dows)?NT/ && 'e/evilandrude.html'
 || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
 || /Linux/ && 'l/Linux.html'
 || /HP-UX/ && 'h/HP-SUX.html'
 || /SunOS/ && 's/ScumOS.html'
 || 'a/AppendixB.html';
}
print "Location: $dir/$page\015\012\015\012";
```

Ce type d'instruction `switch` ne fonctionne que lorsque vous savez que les clauses `&&` seront vraies. Si vous ne le savez pas, l'exemple précédent utilisant `?:` devrait être utilisé.

Vous pourriez aussi envisager d'écrire un hachage de références de sous-programmes au lieu de synthétiser une instruction `switch`.

### 27.1.11 Goto

Bien que cela ne soit pas destiné aux âmes sensibles, Perl supporte une instruction `goto`. Il en existe trois formes : `goto-LABEL`, `goto-EXPR`, et `goto-&NAME`. Un `LABEL` de boucle n'est pas en vérité une cible valide pour un `goto`; c'est juste le nom de la boucle.

La forme `goto-LABEL` trouve l'instruction marquée par `LABEL` et reprend l'exécution à cet endroit. Elle ne peut pas être utilisée pour aller dans une structure qui nécessite une initialisation, comme un sous-programme ou une boucle `foreach`. Elle ne peut pas non plus être utilisée pour aller dans une structure très optimisée. Elle peut être employée pour aller presque n'importe où ailleurs à l'intérieur de la portée dynamique, y compris hors des sous-programmes, mais il est habituellement préférable d'utiliser une autre construction comme `last` ou `die`. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl, à vrai dire - C est une toute autre question).

La forme `goto-EXPR` attend un nom de label, dont la portée sera résolue dynamiquement. Ceci permet des `gotos` calculés à la mode de FORTRAN, mais ce n'est pas nécessairement recommandé si vous optimisez la maintenance du code :

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

La forme `goto-&NAME` est hautement magique, et substitue au sous-programme en cours d'exécution un appel au sous-programme nommé. C'est utilisé par les sous-programmes `AUTOLOAD()` qui veulent charger une autre routine et prétendre que cette autre routine a été appelée à leur place (sauf que toute modification de `@_` dans le sous-programme en cours est propagée à l'autre routine). Après le `goto`, même `caller()` ne pourra pas dire que cette routine n'a pas été appelée en premier.

Dans presque tous les cas similaires, une bien, bien meilleure idée est d'utiliser les mécanismes de contrôle de flux structurés comme `next`, `last`, ou `redo` au lieu de s'en remettre à un `goto`. Pour certaines applications, la paire `eval{} - die()` pour le traitement des exceptions peut aussi être une approche prudente.

### 27.1.12 POD : documentation intégrée

Perl dispose d'un mécanisme pour mélanger de la documentation avec le code source. Lorsqu'il attend le début d'une nouvelle instruction, si le compilateur rencontre une ligne commençant par un signe égal et un mot, comme ceci

```
=head1 Here There Be Pods!
```

Alors ce texte et tout ce qui suit jusqu'à et y compris une ligne commençant par `=cut` sera ignoré. Le format du texte en faisant partie est décrit dans *perlpod*.

Ceci vous permet de mélanger librement votre code source et votre documentation, comme dans

```
=item snazzle($)
```

La fonction `snazzle()` se comportera de la façon la plus spectaculaire que vous pouvez imaginer, y compris la pyrotechnie cybernétique.

```
=cut retour au compilateur, nuff of this pod stuff!
```

```
sub snazzle($) {
 my $thingie = shift;

}
```

Notez que les traducteurs `pod` ne devraient traiter que les paragraphes débutant par une directive `pod` (cela rend leur analyse plus simple), tandis que le compilateur sait en réalité chercher des séquences `pod` même au milieu d'un paragraphe. Cela signifie que le bout de code secret qui suit sera ignoré à la fois par le compilateur et les traducteurs.

```
$a=3;
=truc secret
 warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

Vous ne devriez probablement pas vous reposer sur le fait que le `warn()` sera ignoré pour toujours. Les traducteurs `pod` ne sont pas tous bien élevés de ce point de vue, et le compilateur deviendra peut-être plus regardant.

On peut aussi utiliser des directives `pod` pour mettre rapidement une partie de code en commentaire.

### 27.1.13 Bons Vieux Commentaires (Non !)

Perl peut traiter des directives de ligne, à la manière du préprocesseur C. Avec cela, on peut contrôler l'idée que Perl se fait des noms de fichiers et des numéros de ligne dans les messages d'erreur ou dans les avertissements (en particulier pour les chaînes traitées par `eval()`). La syntaxe de ce mécanisme est la même que pour la plupart des préprocesseurs C : elle reconnaît l'expression régulière :

```
example: '# line 42 "new_filename.plx"'
/^\# \s*
 line \s+ (\d+) \s*
 (?:\s("?)([\^"]+)\2)? \s*
$/x
```

avec \$1 qui est le numéro de ligne pour la ligne suivante et \$3 qui est le nom optionnel du fichier (avec ou sans guillemets). Voici quelques exemples que vous devriez pouvoir taper dans votre interpréteur de commandes :

```
% perl
line 200 "bzzzt"
the '#' on the previous line must be the first char on line
die 'foo';
__END__
foo at bzzzt line 201.
```

```
% perl
line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.
```

```
% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.
```

```
% perl
line 345 "goop"
eval "\n#line " . __LINE__ . ' ' . __FILE__ . "\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

## 27.2 TRADUCTION

### 27.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 27.2.2 Traducteur

Traduction initiale : Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 27.2.3 RELECTURE

Régis Julié <[Regis.Julie@cetelem.fr](mailto:Regis.Julie@cetelem.fr)>, Etienne Gauthier <[egauthie@capgemini.fr](mailto:egauthie@capgemini.fr)>

# Chapitre 28

## perldata

Types de données de Perl

### 28.1 DESCRIPTION

#### 28.1.1 Noms des variables

Perl a trois types de données intégrés : les scalaires, les tableaux de scalaires et les tableaux associatifs de scalaires, appelés aussi tables de hachage ou « hachages ». Un scalaire est soit une simple chaîne de caractères (sans limite de taille si ce n'est la mémoire disponible), soit un nombre, soit une référence à quelque chose (notion expliquée dans *perlref*). Les tableaux normaux sont des listes ordonnées de scalaires indexées par des nombres, en commençant par 0. Les tables de hachages sont des collections non ordonnées de valeurs scalaires indexées par des chaînes qui sont leurs clés associées.

On fait habituellement référence aux valeurs par leur nom, ou par une référence nommée. Le premier caractère du nom vous indique à quel type de structure de données il correspond. Le reste du nom vous dit à quelle valeur particulière il fait référence. Habituellement, ce nom est un simple *identifiant*, c'est-à-dire une chaîne commençant par une lettre ou un caractère souligné, et contenant des lettres, des soulignés, et des chiffres. Dans certains cas, il peut être une chaîne d'identifiants, séparés par `::` (ou par le légèrement archaïque `'`); tous sauf le dernier sont interprétés comme des noms de paquetages, pour localiser l'espace de nommage dans lequel l'identifiant final doit être recherché (voir *Paquetages in perlmod* pour plus de détails). Il est possible de substituer à un simple identifiant une expression qui produit une référence à la valeur lors de l'exécution. Ceci est décrit plus en détails plus bas, et dans *perlref*.

Perl a aussi ses propres variables intégrées dont les noms ne suivent pas ces règles. Elles ont des noms étranges pour qu'elles ne rentrent pas accidentellement en collision avec l'une de vos variables normales. Les chaînes qui correspondent aux parties entre parenthèses d'une expression rationnelle sont sauvées sous des noms qui ne contiennent que des chiffres après le `$` (voir *perlop* et *perlre*). De plus, plusieurs variables spéciales qui ouvrent des fenêtres dans le fonctionnement interne de Perl ont des noms contenant des signes de ponctuation et des caractères de contrôle. Elles sont décrites dans *perlvar*.

Les valeurs scalaires sont toujours désignées par un `'$'`, même si l'on se réfère à un scalaire qui fait partie d'un tableau ou d'un hachage. Le symbole `'$'` fonctionne d'un point de vue sémantique comme les articles « le » ou « la » dans le sens où ils indiquent qu'on attend une seule valeur.

```
$days # la simple valeur scalaire "days"
$days[28] # le 29e élément du tableau @days
$days{'Feb'} # la valeur 'Feb' dans le hachage %days
$#days # le dernier indice du tableau @days
```

Les tableaux complets (et les tranches de tableaux ou de hachages) sont dénotés par `'@'`, qui fonctionne plutôt comme le mot « ces », en ce sens qu'il indique que de multiples valeurs sont attendues .

```
@days # ($days[0], $days[1],... $days[n])
@days[3,4,5] # identique à ($days[3], $days[4], $days[5])
@days{'a','c'} # identique à ($days{'a'}, $days{'c'})
```

Les hachages complets sont dénotés par `'%'` :

```
%days # (clé1, valeur1, clé2, valeur2 ...)
```

De plus, les sous-programmes sont nommés avec un '&' initial, bien que ce soit optionnel lorsqu'il n'y a pas d'ambiguïté, tout comme « faire » est souvent redondant en français. Les entrées des tables de symboles peuvent être nommées avec un '\*' initial, mais vous ne vous souciez pas vraiment de cela pour le moment (sait-on jamais... :-)).

Chaque type de variable a son propre espace de nommage, tout comme les identifiants de plusieurs types autres que les variables. Ceci signifie que vous pouvez, sans craindre de conflit, utiliser le même nom pour une variable scalaire, un tableau, ou un hachage – mais aussi pour un handle de fichier, un handle de répertoire, un nom de sous-programme, ou un label. Ceci veut dire que \$foo et @foo sont deux variables différentes. Ceci veut aussi dire que \$foo[1] fait partie de @foo, et pas de \$foo. Cela peut sembler un peu étrange, mais c'est normal, puisque c'est étrange.

Puisque les références de variables commencent toujours par '\$', '@', ou '%', les mots « réservés » ne sont en fait pas réservés en ce qui concerne les noms de variables (Ils SONT toutefois réservés en ce qui concerne les labels et les handles de fichiers, qui n'ont pas de caractère spécial initial. Vous ne pouvez pas avoir un handle de fichier nommé « log », par exemple. Indice : vous pourriez dire `open(LOG, 'logfile')` plutôt que `open(log, 'logfile')`. Utiliser des handles de fichiers en lettres majuscules améliore aussi la lisibilité et vous protège de conflits avec de futurs mots réservés. La casse *est* significative – « FOO », « Foo », et « foo » sont tous des noms différents. Les noms qui commencent par une lettre ou un caractère souligné peuvent aussi contenir des chiffres et des soulignés.

Il est possible de remplacer un tel nom alphanumérique par une expression qui retourne une référence au type approprié. Pour une description de ceci, voir *perlref*.

Les noms qui commencent par un chiffre ne peuvent contenir que des chiffres. Les noms qui ne commencent pas par une lettre, un souligné, un chiffre ou un accent circonflexe (donc un caractère de contrôle) sont limités à un caractère, tels \$% or \$\$ (La plupart de ces noms d'un seul caractère ont une signification prédéfinie pour Perl. Par exemple, \$\$ est l'ID du processus courant).

## 28.1.2 Contexte

L'interprétation des opérations et des valeurs en Perl dépend parfois des exigences du contexte de l'opération ou de la valeur. Il existe deux contextes majeurs : le contexte de liste et le contexte scalaire. Certaines opérations retournent des valeurs de liste dans les contextes qui réclament une liste, et des valeurs scalaires autrement. Si ceci est vrai pour une opération alors cela sera mentionné dans la documentation pour cette opération. En d'autres termes, Perl surcharge certaines opérations selon que la valeur de retour attendue est singulière ou plurielle. Certains mots en français fonctionnent aussi de cette façon, comme « lys » et « dos ».

Réciproquement, une opération fournit un contexte scalaire ou de liste à chacun de ses arguments. Par exemple, si vous dites :

```
int(<STDIN>)
```

L'opération `int` fournit un contexte scalaire pour l'opérateur `<STDIN>`, qui répond en lisant une ligne depuis `STDIN` et en la passant à l'opération `int`, qui trouvera alors la valeur entière de cette ligne et retournera cela. Si, au contraire, vous dites :

```
sort(<STDIN>)
```

alors l'opération `sort` fournit un contexte de liste pour `<STDIN>`, qui se mettra à lire toutes les lignes disponibles jusqu'à la fin du fichier, et passera cette liste de lignes à la routine de tri, qui triera alors ces lignes et les retournera en tant que liste à ce qui est le contexte de `sort`, quel qu'il soit.

L'affectation est un petit peu spéciale en ce sens qu'elle utilise son argument gauche pour déterminer le contexte de l'argument droit. L'affectation à un scalaire évalue la partie droite dans un contexte scalaire, tandis que l'affectation à un tableau ou à un hachage évalue la partie droite dans un contexte de liste. L'affectation à une liste (ou à une tranche, qui est juste une liste de toute façon) évalue aussi la partie droite dans un contexte de liste.

Lorsque vous utilisez le pragma `use warnings` ou l'option de ligne de commande `-w` de Perl, il arrive que vous voyiez des avertissements sur un usage inutile de constantes ou de fonctions dans un « contexte vide » (« void context », NDT). Le contexte vide signifie juste que la valeur a été abandonnée, comme pour une instruction ne contenant que "fred"; ou `getpwuid(0)`; . Il compte toujours pour un contexte scalaire pour les fonctions qui se soucient de savoir si elles sont ou non appelées dans un contexte scalaire.

Les sous-programmes définis par l'utilisateur peuvent se soucier d'avoir été appelés dans un contexte vide, scalaire ou de liste. La plupart des sous-programmes n'en ont toutefois pas besoin. C'est parce que les scalaires et les listes sont automatiquement interpolés en listes. Voir `wantarray()` dans *perlfunc* pour une façon dont vous pourriez discerner dynamiquement le contexte d'appel de votre fonction.

### 28.1.3 Valeurs scalaires

Toute donnée en Perl est un scalaire, un tableau de scalaires ou un hachage de scalaires. Les variables scalaires peuvent contenir des une seule valeur de trois formes différentes : un nombre, une chaîne ou une référence. En général, la conversion d'une forme à une autre est transparente. Bien qu'un scalaire ne puisse pas contenir des valeurs multiples, il peut contenir une référence à un tableau ou à un hachage qui à son tour contient des valeurs multiples.

Les scalaires ne sont pas nécessairement une chose ou une autre. Il n'y a pas d'endroit où déclarer qu'une variable scalaire doit être de type « chaîne », de type « nombre », de type « référence », ou n'importe quoi d'autre. Du fait de la conversion automatique des scalaires, les opérations qui en retournent n'ont pas besoin de se soucier (et en fait ne le peuvent pas) de savoir si leur appelant attend une chaîne, un nombre ou une référence. Perl est un langage contextuellement polymorphe dont les scalaires peuvent être des chaînes, des nombres, ou des références (ce qui inclut les objets). Tandis que les chaînes et les nombres sont considérés comme presque la même chose pour pratiquement tous les usages, les références sont des pointeurs au typage fort et impossible à forcer, avec comptage de référence intégré et invocation de destructeur.

Une valeur scalaire est interprétée comme TRUE (VRAIE, NDT) au sens booléen si ce n'est pas une chaîne vide ou le nombre 0 (ou son équivalent sous forme de chaîne, « 0 »). Le contexte booléen est juste un genre spécial de contexte scalaire, où aucune conversion vers une chaîne ou un nombre n'est jamais effectuée.

Il existe en fait deux variétés de chaînes nulles (parfois appelées des chaînes « vides »), l'une définie et l'autre non. La version définie est juste une chaîne de longueur zéro, telle que "". La version non définie est la valeur qui indique qu'il n'existe pas de vraie valeur pour quelque chose, comme lorsqu'il s'est produit une erreur, ou à la fin d'un fichier, ou lorsque vous vous référez à une variable ou à un élément de tableau ou de hachage non initialisé. Bien que dans les anciennes versions de Perl, un scalaire indéfini ait pu devenir défini lorsqu'il était utilisé pour la première fois dans un endroit où une valeur définie était attendue, cela ne se produit plus, sauf dans de rares cas d'autovivification tels qu'expliqués dans *perlref*. Vous pouvez utiliser l'opérateur `defined()` pour déterminer si une valeur scalaire est définie (cela n'a pas de sens pour les tableaux ou les hachages), et l'opérateur `undef()` pour produire une valeur indéfinie.

Pour trouver si une chaîne donnée est un nombre différent de zéro valide, il suffit parfois de la tester à la fois avec le 0 numérique et le « 0 » lexical (bien que ceci provoquera du bruit si les avertissements sont actifs). C'est parce que les chaînes qui ne sont pas des nombres comptent comme 0, tout comme en **awk** :

```
if ($str == 0 && $str ne "0") {
 warn "That doesn't look like a number";
}
```

Cette méthode est peut-être meilleure parce qu'autrement vous ne traiteriez pas correctement les notations IEEE comme NaN ou Infinity. À d'autres moments, vous pourriez préférer déterminer si une donnée chaîne peut être utilisée numériquement en appelant la fonction `POSIX::strtod()` ou en inspectant votre chaîne avec une expression rationnelle (tel que documenté dans *perlre*).

```
warn "has nondigits" if /\D/;
warn "not a whole number" unless /\d+$/;
warn "not an integer" unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^[+-]?\d+\.\d*$/;
warn "not a C float"
 unless /^[+-]?(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$;/;
```

La longueur d'un tableau est une valeur scalaire. Vous pourriez trouver la longueur du tableau `@days` en évaluant `$#days`, comme en **csh**. Techniquement, ce n'est pas la longueur du tableau ; c'est l'indice de son dernier élément, ce qui n'est pas la même chose parce qu'il y a habituellement un élément numéro 0. Une affectation à `$#days` change véritablement la longueur du tableau. Le raccourcissement d'un tableau par cette méthode détruit les valeurs intermédiaires. L'agrandissement d'un tableau ayant précédemment été raccourci ne récupère pas les valeurs qui étaient stockées dans ces éléments (c'était le cas en Perl 4, mais nous avons dû supprimer cela pour nous assurer que les destructeurs sont bien appelés quand on s'y attend).

Vous pouvez aussi gagner un tout petit peu d'efficacité en pré-étendant un tableau qui va devenir gros (vous pouvez aussi étendre un tableau en affectant des données à un élément qui est au-delà de la fin du tableau). Vous pouvez tronquer totalement un tableau en y affectant la liste vide `()`. Les expressions suivantes sont équivalentes :

```
@whatever = ();
$#whatever = -1;
```

Si vous évaluez un tableau dans un contexte scalaire, cela renvoie la longueur du tableau (notez que ceci n'est pas vrai pour les listes, qui renvoient leur dernière valeur, comme l'opérateur virgule en C, et contrairement aux fonctions intégrées, qui renvoient ce qu'elles ont envie de renvoyer). Ce qui suit est toujours vrai :

```
scalar(@whatever) == $#whatever - $[+ 1;
```

La version 5 de Perl a changé la sémantique de `$[` : les fichiers qui ne fixent pas la valeur de `$[` n'ont plus besoin de s'inquiéter de savoir si un autre fichier a changé sa valeur (en d'autres termes, l'usage de `$[` est désapprouvé). Donc de façon générale, vous pouvez présumer que

```
scalar(@whatever) == $#whatever + 1;
```

Certains programmeurs choisissent d'utiliser une conversion explicite pour ne rien laisser au hasard :

```
$element_count = scalar(@whatever);
```

Si vous évaluez un hachage dans un contexte scalaire, vous obtenez faux si le hachage est vide. S'il contient une paire clé/valeur quelconque, il renvoie vrai ; plus précisément, la valeur retournée est une chaîne constituée du nombre de buckets utilisés et du nombre de buckets alloués, séparés par un signe de division. Ceci n'a d'intérêt que pour déterminer si les algorithmes de hachage (compilés) en Perl ont des performances médiocres sur vos données. Par exemple, vous mettez 10 000 trucs dans un hachage, mais l'évaluation de `%HASH` dans un contexte scalaire révèle « 1/16 », ce qui signifie qu'un seul des seize buckets a été touché, et contient probablement tous vos 10 000 éléments. Cela ne devrait pas se produire. L'évaluation d'une table de hachage liée (par 'tie') dans un contexte scalaire engendrera une erreur fatale puisque l'information d'utilisation de bucket pour les tables de hachage liées n'est pour l'instant pas disponible.

Vous pouvez préallouer de la mémoire pour une table de hachage en affectant une valeur à la fonction `keys()`. Ceci arrondi le nombre de buckets alloués à la puissance de deux directement supérieure :

```
keys(%users) = 1000; # pré-alloue 1024 buckets
```

## 28.1.4 Constructeurs de valeurs scalaires

Les littéraux numériques sont spécifiés dans un quelconque des formats suivants de nombres entiers ou à virgule flottante :

```
12345
12345.67
.23E-10 # un très petit nombre
3.14_15_92 # un nombre très important
4_294_967_296 # souligné pour la lisibilité
0xff # hexa
0xfade_bebe # un autre hexa
0377 # octal (des chiffres commençant par 0)
0b011011 # binaire
```

Vous pouvez utiliser le caractère underscore (souligné) dans un nombre littéral entre les chiffres pour améliorer la lisibilité. Vous pouvez, par exemple, regrouper les chiffres par trois (pour les droits d'accès Unix tel `0b110_100_100`) ou par quatre (pour représenter des chiffres hexa dans `0b1010_0110`) ou tout autre regroupement.

Les littéraux de chaîne sont habituellement délimités soit par des apostrophes, soit par des guillemets. Ils fonctionnent beaucoup comme dans un shell Unix standard : les littéraux de chaîne entre guillemets sont sujets aux substitutions de variables et au préfixage par barre oblique inverse ; les chaînes entre apostrophes ne le sont pas (sauf pour `\'` et `\\`). Les règles habituelles d'utilisation de la barre oblique inverse en C s'appliquent aussi bien pour créer des caractères comme la nouvelle ligne, la tabulation, etc., que sous des formes plus exotiques. Voir *Opérateurs apostrophe et type apostrophe* in *perlop* pour une liste.

Les représentations hexadécimales, octales ou binaires sous forme de chaînes (e.g. `'0xff'`) ne sont pas automatiquement converties sous leur représentation entière. Les fonctions `hex()` et `oct()` font ces conversions pour vous. Voir *hex()* et *oct()* pour plus de détails.

Vous pouvez aussi inclure des « nouvelles lignes » directement dans vos chaînes, i.e., elles peuvent se terminer sur une ligne différente de celles où elles ont commencé. C'est bien joli, mais si vous oubliez votre apostrophe de fin (ou votre guillemet - NDT), l'erreur ne sera pas rapportée avant que Perl n'ait trouvé une autre ligne comportant une apostrophe, qui peut se trouver bien plus loin dans le script. La substitution de variable à l'intérieur des chaînes est limitée aux variables scalaires, aux tableaux et aux tranches de tableau ou de hachage (en d'autres termes, des noms commençant par `$` ou `@`, suivi d'une expression optionnelle entre crochets comme indice). Le segment de code qui suit affiche « The price is \$100. »



```
$Price = '$100'; # pas d'interpolation
print "The price is $Price.\n"; # interpolé
```

Il n'y a pas de double interpolation en Perl, donc '\$100' restera tel quel.

Par défaut les nombres flottants interpolés dans une chaîne utilise le point (".") comme séparateur décimal. Si `use locale` est actif et que `POSIX::setlocale()` a été appelé, le caractère utilisé comme séparateur décimal dépend du locale `LC_NUMERIC`. Voir *perllocale* et *POSIX*.

Comme dans certains shells, vous pouvez mettre des accolades autour d'un nom pour le séparer des caractères alphanumériques (et du caractère souligné) qui le suivent. Vous devez aussi faire cela lorsque vous interpolatez une variable dans une chaîne pour séparer son nom d'un deux-points ou d'une apostrophe, puisqu'ils seraient autrement traités comme un séparateur de paquetage :

```
$who = "Larry";
print PASSWD "${who}::0:0:Superuser:~/bin/perl\n";
print "We use ${who}speak when ${who}'s here.\n";
```

Sans les accolades, Perl aurait cherché un `$whospeak`, un `$who::0`, et une variable `who's`. Les deux dernières auraient été les variables `$0` et `$s` dans le paquetage `who` (probablement) inexistant.

En fait, un identifiant situé entre de telles accolades est forcé d'être une chaîne, tout comme l'est tout identificateur isolé à l'intérieur d'un indice d'un hachage. Aucun des deux n'a besoin d'apostrophes. Notre exemple précédent, `$days{'Feb'}` peut être écrit sous la forme `$days{Feb}` et les apostrophes seront présumées automatiquement. Mais tout ce qui est plus compliqué dans l'indice sera interprété comme étant une expression. Cela signifie par exemple que `$version{2.0}` sera interprété comme `$version{2}` et non comme `$version{'2.0'}`.

### Chaîne de version

**NOTE** : les chaînes de version (vstrings) sont dépréciées. Elle seront supprimées dans l'une des versions qui suivront Perl 5.8.1. Le bénéfice minime des chaînes de version est largement inférieur aux mauvaises surprises et confusions potentielles qu'elles entraînent.

Un littéral de la forme `v1.20.300.4000` est analysé comme une chaîne composée de caractères correspondants aux ordinaux spécifiés. Cette formulation, connu sous le nom v-string ou chaîne de version, fournit une façon plus lisible de construire des chaînes, au lieu d'utiliser l'interpolation parfois un peu moins lisible `"\x{1}\x{14}\x{12c}\x{fa0}"`. C'est utile pour représenter des chaînes Unicode, et pour comparer des numéros de version en utilisant les opérateurs de comparaison de chaînes, `cmp`, `gt`, `lt`, etc. Si le littéral contient plusieurs points, le `v` initial peut être omis.

```
print v9786; # affiche le SMILEY codé en UTF-8,
 # "\x{263a}"
print v102.111.111; # affiche "foo"
print 102.111.111; # idem
```

De tels littéraux sont acceptés à la fois par `require` et `use` pour réaliser une vérification de numéro de version. La variable spéciale `$^V` contient aussi le numéro de version sous cette forme de l'interpréteur Perl en cours d'utilisation. Voir `$^V` in *perlvar*. Notez que l'utilisation des chaînes de version pour construire des adresses IPv4 n'est pas portable à moins d'utiliser les fonctions `inet_aton()` et `inet_ntoa()` du package `Socket`.

Notez que depuis Perl 5.8.1, une chaîne de version à un seul nombre (comme `v65`) n'en est pas une si elle est avant l'opérateur `=>` (qui est utilisé habituellement pour séparer une clé d'une valeur dans une table de hachage). C'était considéré comme une chaîne de version depuis Perl 5.6.0 jusqu'à Perl 5.8.0 mais cela a provoqué plus de confusion et d'erreurs que d'avantages. En revanche, les chaînes de version multi-nombres comme `v65.66` et `65.66.67` le restent bien.

### Littéraux spéciaux

Les littéraux spéciaux `__FILE__`, `__LINE__`, et `__PACKAGE__` représentent le nom de fichier courant, le numéro de la ligne, et le nom du paquetage à ce point de votre programme. Ils ne peuvent être utilisés que comme des mots-clé isolés ; ils ne seront pas interpolés dans les chaînes. S'il n'existe pas de paquetage courant (à cause d'une directive `package;`), `__PACKAGE__` est la valeur indéfinie.

Les deux caractères de contrôle `^D` et `^Z`, et les mots-clé `__END__` et `__DATA__` peuvent être utilisés pour indiquer la fin logique d'un script avant la fin effective du fichier. Tout texte les suivant est ignoré.

Le texte qui suit `__DATA__` peut être lu via le handle de fichier `PACKNAME::DATA`, où `PACKNAME` est le paquetage qui était courant lorsque le mot-clé `__DATA__` a été rencontré. Le handle de fichier est laissé ouvert, pointant vers le contenu après `__DATA__`. Il est de la responsabilité du programme d'effectuer un `close DATA` lorsqu'il a fini d'y lire. Pour la compatibilité avec d'anciens scripts écrits avant que `__DATA__` ne soit introduit, `__END__` se comporte comme `__DATA__` dans le script principal (mais pas dans les fichiers chargés par `require` ou `do`) et laisse le contenu restant du fichier accessible via `main::DATA`.

Voir *SelfLoader* pour une plus longue description de `__DATA__`, et un exemple de son utilisation. Notez que vous ne pouvez pas lire depuis le handle de fichier `DATA` dans un bloc `BEGIN` : ce bloc est exécuté dès qu'il est vu (pendant la compilation), à un moment où le mot-clé `__DATA__` (ou `__END__`) correspondant n'a pas encore été rencontré.

### Mots simples (ou « barewords »)

Un mot qui n'a aucune autre interprétation dans la grammaire sera traité comme s'il était une chaîne entre apostrophes. Ces mots sont connus sous le nom de « barewords ». Comme pour les handles de fichier et les labels, un bareword constitué entièrement de lettres minuscules risque d'entrer en conflit avec de futurs mots réservés, et si vous utilisez le pragma `use warnings` ou l'option `-w`, Perl vous avertira pour chacun d'entre eux. Certaines personnes pourraient vouloir rendre les barewords totalement hors-la-loi. Si vous dites

```
use strict 'subs';
```

alors tout bareword qui ne serait PAS interprété comme un appel à un sous-programme produit à la place une erreur au moment de la compilation. La restriction continue jusqu'à la fin du bloc qui le contient. Un bloc interne pourrait annuler ceci en disant `no strict 'subs'`.

### Délimiteur de jointure des tableaux

Les tableaux et les tranches sont interpolés dans les chaînes entre guillemets en joignant tous les éléments avec le délimiteur spécifié dans la variable `$` (`$LIST_SEPARATOR` si "use English" est spécifié), un espace par défaut. Les expressions suivantes sont équivalentes :

```
$temp = join("$", @ARGV);
system "echo $temp";
```

```
system "echo @ARGV";
```

À l'intérieur d'un motif de recherche (qui subit aussi la substitution entre guillemets) il y a une malheureuse ambiguïté : est-ce que `/$foo[bar]/` doit être interprété comme `/${foo}[bar]/` (où `[bar]` est une classe de caractères pour l'expression régulière) ou comme `/${foo[bar]}/` (où `[bar]` est un indice du tableau `@foo`) ? Si `@foo` n'existe pas par ailleurs, alors c'est évidemment une classe de caractères. Si `@foo` existe, Perl choisit de deviner la valeur de `[bar]`, et il a presque toujours raison. S'il se trompe, ou si vous êtes simplement complètement paranoïaque, vous pouvez forcer l'interprétation correcte avec des accolades comme ci-dessus.

Si vous cherchez comment utiliser les here-documents, les informations qui étaient précédemment ici sont maintenant dans *Opérateurs apostrophe et type apostrophe* in *perlop*.

## 28.1.5 Constructeurs de listes de valeurs

Les valeurs de liste sont notées en séparant les valeurs individuelles par des virgules (et en enfermant la liste entre parenthèses lorsque la précédence le requiert) :

```
(LIST)
```

Dans un contexte qui ne requiert pas une valeur de liste, la valeur de ce qui apparaît être un littéral de liste est simplement la valeur de l'élément final, comme avec l'opérateur virgule en C. Par exemple,

```
@foo = ('cc', '-E', $bar);
```

affecte la totalité de la valeur de liste au tableau `@foo`, mais

```
$foo = ('cc', '-E', $bar);
```

affecte la valeur de la variable \$bar à la variable \$foo. Notez que la valeur d'un véritable tableau dans un contexte scalaire est la longueur du tableau ; ce qui suit affecte la valeur 3 à \$foo :

```
@foo = ('cc', '-E', $bar);
$foo = @foo; # $foo prend la valeur 3
```

Vous pouvez avoir une virgule optionnelle avant la parenthèse fermante d'un littéral de liste, vous pouvez donc dire :

```
@foo = (
 1,
 2,
 3,
);
```

Pour utiliser un here-document afin d'affecter un tableau, une ligne par élément, vous pourriez utiliser l'approche suivante :

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
 normal tomato
 spicy tomato
 green chile
 pesto
 white wine
End_Lines
```

Les LIST font une interpolation automatique des sous-listes. C'est-à-dire que lorsqu'une LIST est évaluée, chaque élément de la liste est évalué dans un contexte de liste, et la valeur de liste résultante est interpolée en LIST tout comme si chaque élément était un membre de LIST. Ainsi, les tableaux perdent leur identité dans une LIST - la liste

```
(@foo,@bar,&SomeSub)
```

contient tous les éléments de @foo suivis par tous les éléments de @bar, suivis par tous les éléments retournés par le sous-programme appelé SomeSub quand il est appelé dans un contexte de liste. Pour faire une référence à une liste qui NE soit PAS interpolée, voir *perlref*.

La liste vide est représentée par (). L'interpoler dans une liste n'a aucun effet. Ainsi, ((),(),()) est équivalent à (). De façon similaire, interpoler un tableau qui ne contient pas d'élément est exactement comme si aucun tableau n'est interpolé à cet endroit-là.

L'interpolation combine l'optionnalité des parenthèses ouvrantes et fermantes (sauf quand la précedence l'impose) et la possibilité de terminer une liste par une virgule optionnelle pour accepter comme syntaxiquement légal plusieurs virgules successives dans une liste. La liste 1,,3 est la concaténation des deux listes 1, et 3 et la première possède une virgule optionnelle. 1,,3 est (1, ), (3) et donc 1, 3 (c'est la même chose pour 1,, 3 qui est (1, ), (, ), 3 et donc 1, 3 et ainsi de suite). Ceci n'est pas un encouragement vers l'illisibilité...

Une valeur de liste peut aussi être indiquée comme un tableau normal. Vous devez mettre la liste entre parenthèses pour éviter les ambiguïtés. Par exemple :

```
Stat renvoie une valeur de liste.
$time = (stat($file))[8];

ICI, ERREUR DE SYNTAXE.
$time = stat($file)[8]; # OOPS, OUBLI DES PARENTHESES

Trouver un chiffre hexadécimal.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

Un "opérateur virgule inversé".
return (pop(@foo),pop(@foo))[0];
```

Les listes ne peuvent être affectées que si chaque élément de la liste peut l'être lui aussi :

```
($a, $b, $c) = (1, 2, 3);
```

```
($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

Une exception à ceci est que vous pouvez affecter undef dans une liste. C'est pratique pour balancer certaines valeurs de retour d'une fonction :

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

L'affectation de liste dans un contexte scalaire renvoie le nombre d'éléments produits par l'expression du côté droit de l'affectation :

```
$x = (($foo,$bar) = (3,2,1)); # met 3 dans $x, pas 2
$x = (($foo,$bar) = f()); # met le nombre de valeurs
 # de retour de f() dans $x
```

Ceci est pratique lorsque vous voulez faire une affectation de liste dans un contexte booléen, parce que la plupart des fonctions de liste renvoient une liste vide quand elle se terminent, ce qui donne un 0 quand on l'affecte, 0 qui est interprété comme FALSE (FAUX - NDT).

Ça fournit aussi un moyen idiomatique d'exécuter une fonction ou une opération dans un contexte de liste puis de compter le nombre de valeurs retournées, en assignant le résultat à une liste vide puis en l'affectant dans un contexte scalaire. Par exemple, le code suivant :

```
$count = () = $string =~ /\d+/g;
```

placera dans \$count le nombre de groupes de chiffres trouvés dans \$string. Cela se passe comme ça parce que la recherche de motifs a lieu dans un contexte de liste (puisqu'on l'affecte à la liste vide) et retournera donc la liste de toutes les parties reconnues dans la chaîne. L'affectation de la liste dans un contexte scalaire convertira cette liste en son nombre d'éléments (ici, le nombre de reconnaissances du motif) et ce nombre sera affecté à \$count. Remarquez que l'utilisation de :

```
$count = $string =~ /\d+/g;
```

ne fonctionne pas puisque la recherche de motifs est dans un contexte scalaire et retourne donc juste vrai ou faux au lieu du nombre de reconnaissances.

L'élément final d'une liste d'affectation peut être un tableau ou un hachage :

```
($a, $b, @rest) = split;
local($a, $b, %rest) = @_;
```

Vous pouvez en vérité mettre un tableau ou un hachage n'importe où dans la liste, mais le premier situé dans la liste va aspirer toutes les valeurs, et tout ce qui le suivra deviendra indéfini. Cela peut être pratique dans un local() ou un my().

Un hachage peut être initialisé en utilisant une liste de littéraux contenant des paires d'éléments qui doivent être interprétées comme des couples clé/valeur :

```
identique à l'affectation de map ci-dessus
%map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

Tandis que les littéraux de liste et les tableaux nommés sont souvent interchangeable, ce n'est pas le cas pour les hachages. Le simple fait que vous puissiez indiquer une valeur de liste comme un tableau normal ne veut pas dire que vous pouvez indiquer une valeur de liste comme un hachage. De la même manière, les hachages inclus comme parties d'autres listes (y compris les listes de paramètres et les listes de retour de fonctions) s'aplatissent toujours en paires clé/valeur. C'est pourquoi il est parfois bon d'utiliser des références.

Il est parfois plus lisible d'utiliser l'opérateur => dans les paires clé/valeur. L'opérateur => est principalement le synonyme d'une virgule mais visuellement plus clair. Il permet aussi à son opérande de gauche d'être interprété comme une chaîne, si c'est un bareword qui serait un identifiant légal (=> n'agit tout de même pas sur les identifiants composés, ceux qui contiennent des deux-points). Cela rend plus jolie l'initialisation des hachages :

```
%map = (
 red => 0x00f,
 blue => 0x0f0,
 green => 0xf00,
);
```

ou pour initialiser les références de hachage devant être utilisées en tant qu'enregistrements :

```
$rec = {
 witch => 'Mable the Merciless',
 cat => 'Fluffy the Ferocious',
 date => '10/31/1776',
};
```

ou pour utiliser l'appel par variables pour les fonctions compliquées :

```
$field = $query->radio_group(
 name => 'group_name',
 values => ['eenie', 'meenie', 'minie'],
 default => 'meenie',
 linebreak => 'true',
 labels => \%labels
);
```

Notez que ce n'est pas parce qu'un hachage est initialisé dans un certain ordre qu'il ressortira dans cet ordre. Voir `sort()` pour des exemples sur la façon de s'arranger pour obtenir des sorties ordonnées.

### 28.1.6 Indices

L'accès aux éléments d'un tableau s'effectue en écrivant le symbole dollar (\$) puis le nom du tableau (sans le @ initial) puis l'indice de l'élément voulu entre crochets. Par exemple :

```
@tableau = (5, 50, 500, 5000);
print "L'élément numéro 2 est : ", $tableau[2], "\n";
```

Les indices dans les tableaux commencent à 0. Un indice négatif retrouve les éléments en partant de la fin du tableau. Dans notre exemple, `$tableau[-1]` vaut 5000 et `$tableau[-2]` vaut 500.

L'accès aux éléments d'une table de hachage est similaire sauf qu'on utilise des accolades à la place des crochets. Par exemple :

```
%scientifiques =
(
 "Newton" => "Isaac",
 "Einstein" => "Albert",
 "Darwin" => "Charles",
 "Feynman" => "Richard",
);

print "Le prénom de Darwin est ", $scientifiques{"Darwin"}, "\n";
```

### 28.1.7 Tranches

Une façon commune d'accéder à un tableau ou à un hachage est d'en prendre un élément à la fois. Vous pouvez aussi indiquer une liste pour en obtenir un seul élément.

```
$whoami = $ENV{"USER"}; # un élément du hachage
$parent = $ISA[0]; # un élément du tableau
$dir = (getpwnam("daemon"))[7]; # idem, mais avec une liste
```

Une tranche accède à plusieurs éléments d'une liste, d'un tableau ou d'un hachage simultanément en utilisant une liste d'indices. C'est plus pratique que d'écrire les éléments individuellement sous la forme d'une liste de valeurs scalaires séparées.

```
($him, $her) = @folks[0,-1]; # tranche de tableau
@them = @folks[0 .. 3]; # tranche de tableau
($who, $home) = @ENV{"USER", "HOME"}; # tranche de hachage
($uid, $dir) = (getpwnam("daemon"))[2,7]; # tranche de liste
```

Puisque vous pouvez affecter à une liste de variables, vous pouvez aussi affecter à une tranche de tableau ou de hachage.

```
@days[3..5] = qw/Wed Thu Fri/;
@colors{'red', 'blue', 'green'}
 = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1] = @folks[-1, 0];
```

Les affectations précédentes sont exactement équivalents à

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
 = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Puisque changer une tranche change le tableau ou le hachage original dont la tranche est issue, une structure `foreach` altèrera certaines – ou même toutes les – valeurs du tableau ou du hachage.

```
foreach (@array[4 .. 10]) { s/peter/paul/ }

foreach (@hash{qw[key1 key2]}) {
 s/^\s+//; # supprime les espaces au début des éléments
 s/\s+$//; # supprime les espaces à la fin des éléments
 s/(\w+)/\u\L$1/g; # met une majuscule aux mots
}
```

Une tranche d'une liste vide est encore une liste vide. Ainsi :

```
@a = ()[1,0]; # @a n'a pas d'éléments
@b = (@a)[0,1]; # @b n'a pas d'éléments
@c = (0,1)[2,3]; # @c n'a pas d'éléments
```

Mais :

```
@a = (1)[1,0]; # @a a deux éléments
@b = (1,undef)[1,0,2]; # @b a trois éléments
```

Ceci rend aisée l'écriture de boucles qui se terminent lorsqu'une liste nulle est renvoyée :

```
while (($home, $user) = (getpwent)[7,0]) {
 printf "%-8s %s\n", $user, $home;
}
```

Comme noté précédemment dans ce document, le sens scalaire de l'affectation de liste est le nombre d'éléments de la partie droite de l'affectation. La liste nulle ne contient pas d'éléments, donc lorsque le fichier de mots de passe est vidé, le résultat est 0 et non pas 2.

Si vous êtes troublé par le pourquoi de l'usage d'un '@' ici sur une tranche de hachage au lieu d'un '%', pensez-y ainsi. Le type de parenthésage (avec des crochets ou des accolades) décide si c'est un tableau ou un hachage qui est examiné. D'un autre côté, le symbole en préfixe ('\$' ou '@') du tableau ou du hachage indique si vous récupérez une valeur simple (un scalaire) ou une valeur multiple (une liste).

### 28.1.8 Typeglobs et handles de Fichiers

Perl utilise un type interne appelé un *typeglob* pour contenir une entrée complète de table de symbole. Le préfixe de type d'un typeglob est une \*, parce qu'il représente tous les types. Ceci fut la manière favorite de passer par référence à une fonction des tableaux et des hachages, mais maintenant que nous avons de vraies références, c'est rarement nécessaire.

Le principal usage des typeglobs dans le Perl moderne est de créer des alias de table de symbole. Cette affectation :

```
*this = *that;
```

fait de \$this un alias de \$that, @this un alias de @that, %this un alias de %that, &this un alias de &that, etc. Il est bien plus sûr d'utiliser une référence. Ceci :

```
local *Here::blue = \ $There::green;
```

fait temporairement de \$Here::blue un alias de \$There::green, mais ne fait pas de @Hzere::blue un alias de @There::green, ou de %Here::blue un alias de %There::green, etc. Voir Tables de Symboles in *perlmod* pour plus d'exemples de ceci. Aussi étrange que cela puisse paraître, c'est la base de tout le système d'import/export de module.

Un autre usage des typeglobs est le passage de handles de fichiers à une fonction, ou la création de nouveaux handles de fichiers. Si vous avez besoin d'utiliser un typeglob pour sauvegarder un handle de fichier, faites-le de cette façon :

```
$fh = *STDOUT;
```

ou peut-être comme une vraie référence, comme ceci :

```
$fh = *STDOUT;
```

Voir *perlsb* pour des exemples d'usages de ceci comme handles de fichiers indirects dans des fonctions.

Les typeglobs sont aussi une façon de créer un handle de fichier local en utilisant l'opérateur local(). Ceux-ci ne durent que jusqu'à ce que leur bloc soit terminé, mais peuvent être passés en retour. Par exemple :

```
sub newopen {
 my $path = shift;
 local *FH; # not my!
 open (FH, $path) || return undef;
 return *FH;
}
$fh = newopen('/etc/passwd');
```

Maintenant que nous avons la notation \*foo{THING}, les typeglobs ne sont plus autant utilisés pour les manipulations de handles de fichiers, même s'ils sont toujours nécessaires pour passer des fichiers tout neufs et des handles de répertoire dans les fonctions. C'est parce que \*HANDLE{IO} ne fonctionne que si HANDLE a déjà été utilisé en tant que handle. En d'autres termes, \*FH doit être utilisé pour créer de nouvelles entrées de table de symboles ; \*foo{THING} ne le peut pas. En cas de doute, utilisez \*FH.

Toutes les fonctions qui sont capables de créer des handles de fichiers (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), et accept()) créent automatiquement un handle de fichier anonyme si le handle qui leur est passé est une variable scalaire non initialisée. Ceci permet aux constructions telles que open(my \$fh, ...) et open(local \$fh, ...) d'être utilisées pour créer des handles de fichiers qui seront convenablement et automatiquement fermés lorsque la portée se termine, pourvu qu'il n'existe aucune autre référence vers eux. Ceci élimine largement le besoin pour les typeglobs lors de l'ouverture des handles de fichiers qui doivent être passés à droite et à gauche, comme dans l'exemple suivant :

```
sub myopen {
 open my $fh, "@_"
 or die "Can't open '@_': $!";
 return $fh;
}

{
 my $f = myopen("</etc/motd");
 print <$f>;
 # $f implicitly closed here
}
```

Notez que si un scalaire déjà initialisé est utilisé, cela a un effet différent : my \$fh='zzz'; open(\$fh, ...) est équivalent à open(\*{'zzz'}, ...). use strict 'refs' interdit cela.

Une autre façon de créer des handles de fichiers anonymes est d'utiliser le module Symbol ou le module IO::Handle et ceux de son acabit. Ces modules ont l'avantage de ne pas cacher les différents types du même nom pendant le local(). Voir la fin de la documentation de la fonction open() pour un exemple.

## 28.2 VOIR AUSSI

Voir *perlvar* pour une description des variables intégrées de Perl et une discussion des noms de variable légaux. Voir *perlref*, *perlsub*, et Tables de Symboles in *perlmod* pour plus de détails sur les typeglobs et la syntaxe `*foo{THING}`.

## 28.3 TRADUCTION

### 28.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 28.3.2 Traducteur

Traduction initiale : Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit *paul.gaborit at enstimac.fr*.

### 28.3.3 Relecture

Jim Fox <[fox@sugar-land.dowell.slb.com](mailto:fox@sugar-land.dowell.slb.com)>, Etienne Gauthier <[egauthie@capgemini.fr](mailto:egauthie@capgemini.fr)>, Gérard Delafond



# Chapitre 29

## perlop

Opérateurs Perl et priorité

### 29.1 DESCRIPTION

#### 29.1.1 Priorité des opérateurs et associativité

Les règles de priorité et d'associativité des opérateurs en Perl fonctionnent à peu près comme en mathématique.

Les *règles de priorité* entre les opérateurs déterminent l'ordre d'évaluation de ces opérateurs. Par exemple, dans  $2 + 4 * 5$ , la multiplication a une plus grande priorité et donc  $4 * 5$  est évalué en premier pour finalement calculer  $2 + 20 == 22$  et non  $6 * 5 == 30$ .

Les *règles d'associativité* des opérateurs définissent l'ordre dans lequel est évaluée une séquence composée de plusieurs occurrences d'un même opérateur : ces occurrences, selon l'opérateur, peuvent être évaluées en commençant par celle de gauche ou par celle de droite. Par exemple, dans  $8 - 4 - 2$ , la soustraction étant associative à gauche, Perl évalue l'expression de gauche à droite.  $8 - 4$  est évalué en premier pour finalement calculer  $4 - 2 == 2$  et non  $8 - 2 == 6$ .

Le tableau suivant présente les opérateurs Perl et leur priorité, du plus prioritaire au moins prioritaire. Remarquez que tous les opérateurs empruntés au langage C gardent le même ordre de priorité entre eux même lorsque ces priorités sont légèrement tordues. (Cela rend plus simple l'apprentissage de Perl par les programmeurs C.) À quelques exceptions près, tous ces opérateurs agissent sur des valeurs scalaires et pas sur des tableaux de valeurs.

gauche	termes et opérateurs de listes (leftward)
gauche	->
nonassoc	++ --
droite	**
droite	! ~ \ et + et - unaires
gauche	=~ !~
gauche	* / % x
gauche	+ - .
gauche	<< >>
nonassoc	opérateurs nommés unaires
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp
gauche	&
gauche	^
gauche	&&
gauche	//
nonassoc	.. ...
droite	?:
droite	= += -= *= etc.
gauche	, =>
nonassoc	opérateurs de listes (rightward)
droite	not
gauche	and
gauche	or xor err

Dans les sections qui suivent, ces opérateurs sont présentés par ordre de priorité. De nombreux opérateurs peuvent être redéfinis pour des objets. Voir *overload*.

### 29.1.2 Termes et opérateurs de listes (leftward)

Un TERME a la plus haute priorité en Perl. Cela inclut les variables, les apostrophes et autres opérateurs style apostrophe, les expressions entre parenthèses et n'importe quelle fonction dont les arguments sont donnés entre parenthèses. En fait, ce ne sont pas vraiment des fonctions, juste des opérateurs de listes et des opérateurs unaires qui se comportent comme des fonctions parce que vous avez mis des parenthèses autour des arguments. Tout cela est documenté dans *perlfunc*.

Si un opérateur de liste (`print()`, etc.) ou un opérateur unaire (`chdir()`, etc.) est directement suivi par une parenthèse gauche, l'opérateur et les arguments entre parenthèses se voient attribués la priorité la plus haute exactement comme un appel de fonction.

En l'absence de parenthèses, la priorité des opérateurs de liste comme `print`, `sort` ou `chmod` n'est ni très haute ni très basse et dépend de ce qu'il y a à gauche et/ou à droite de l'opérateur. Par exemple, dans :

```
@ary = (1, 3, sort 4, 2);
print @ary; # affiche 1324
```

la virgule à droite du tri (`sort`) est évaluée avant le tri (`sort`) alors que la virgule de gauche est évaluée après. En d'autres termes, un opérateur de listes a tendance à manger tous les arguments qui le suit et à se comporter comme un simple TERME par rapport à l'expression qui le précède. Faites attention aux parenthèses :

```
L'évaluation de exit a lieu avant le print !
print($foo, exit); # Pas vraiment ce que vous vouliez.
print $foo, exit; # Ni dans ce cas.

L'évaluation de print a lieu avant le exit.
(print $foo), exit; # C'est ce que vous voulez.
print($foo), exit; # Ici aussi.
print ($foo), exit; # Et encore dans ce cas.
```

Remarquez aussi que :

```
print ($foo & 255) + 1, "\n";
```

ne donnera probablement pas ce que vous attendiez à priori. Les parenthèses entourent la liste d'arguments du `print` qui est évalué (en affichant le résultat de `$foo & 255`). Ensuite la valeur `un` est additionnée au résultat retourné par `print` (habituellement 1). Le résultat est donc quelque chose comme :

```
1 + 1, "\n"; # Évidemment pas ce que vous souhaitez
```

Pour faire cela proprement, il faut écrire :

```
print(($foo & 255) + 1, "\n");
```

Voir les Opérateurs unaires nommés pour plus de détails.

Sont aussi reconnus comme des termes les constructions `do {}` et `eval {}`, les appels à des sous-routines ou à des méthodes ainsi que les constructeurs anonymes `[]` et `{}`.

Voir aussi Opérateurs apostrophe et type apostrophe à la fin de cette section mais aussi Opérateurs d'E/S (§??).

### 29.1.3 L'opérateur flèche

Comme en C et en C++, "`->`" est un opérateur de déréférencement infixé. Si du côté droit on trouve soit `[...]`, soit `{...}`, soit `(...)` alors le côté gauche doit être une référence vraie ou symbolique vers respectivement un tableau, une table de hachage ou une sous-routine (ou techniquement parlant, un emplacement capable de stocker une référence en dur si c'est une référence vers un tableau ou une table de hachage utilisée pour une affectation). Voir *perlrefut* et *perlref*.

Par contre, si le côté droit est un nom de méthode ou une simple variable scalaire contenant un nom de méthode ou une référence vers une sous-routine alors le côté gauche doit être soit un objet (une référence bénie – par `bless()`) soit un nom de classe (c'est à dire un nom de package). Voir *perlobj*.

### 29.1.4 Auto-incrémentation et auto-décrémentation

Les opérateurs "++" et "--" fonctionnent comme en C. Placés avant la variable, ils incrémentent ou décrémentent la variable avant de retourner sa valeur. Placés après, ils incrémentent ou décrémentent la variable après avoir retourné sa valeur.

```
$i = 0; $j = 0;
print $i++; # affiche 0
print ++$j; # affiche 1
```

Remarquez que, comme en C, Perl ne définit pas **quand** la variable est incrémentée ou décrémentée. Vous savez juste que cela sera fait à un moment avant ou après le moment où la valeur est retournée. Cela signifie aussi que plusieurs modifications de la même variable dans une instruction n'ont pas un comportement garanti. Évitez donc des instructions comme :

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl ne garantit pas le résultat des instructions ci-dessus.

De plus, l'opérateur d'auto-incrémentation inclut un comportement magique. Si vous incrémentez une variable numérique ou qui a déjà été utilisée dans un contexte numérique, vous obtenez l'incrément normal. Si, par contre, la variable n'a été utilisée que dans un contexte de chaîne et correspond au motif `/^[a-zA-Z]*[0-9]*\z/`, l'incrément a lieu sur la chaîne elle-même en préservant les caractères dans leur intervalle et en gérant les éventuelles retenues :

```
print ++($foo = '99'); # affiche '100'
print ++($foo = 'a0'); # affiche 'a1'
print ++($foo = 'Az'); # affiche 'Ba'
print ++($foo = 'zz'); # affiche 'aaa'
```

La valeur `undef` est toujours considérée comme numérique et en particulier est changée en `0` avant son incrément (et donc l'incrément postfixée d'une valeur indéfinie retournera `0` plutôt que `undef`).

L'opérateur d'auto-décrémentation n'est pas magique.

### 29.1.5 Puissance

L'opérateur binaire "\*\*" est l'opérateur puissance. Remarquez qu'il est plus prioritaire que le moins unaire donc `-2**4` signifie `-(2**4)` et non pas `(-2)**4`. (Il est implémenté par la fonction C `pow(3)` qui travaille réellement sur des doubles en interne.)

### 29.1.6 Opérateurs symboliques unaires

L'opérateur unaire "!" est la négation logique, c.-à-d. "non". Voir aussi `not` pour une version moins prioritaire de cette opération.

L'opérateur unaire "-" est la négation arithmétique si l'opérande est numérique. Si l'opérande est un identificateur, il retourne une chaîne constituée du signe moins suivi de l'identificateur. Si la chaîne qui suit commence par un plus ou un moins, la valeur retournée est la chaîne commençant par le signe opposé. L'un des effets de ces règles est que `-bareword` est équivalent à `"-bareword"`. En revanche, si la chaîne qui suit commence par un caractère non-alphabétique (sauf "+" ou "-"), Perl tentera de convertir cette chaîne en une valeur numérique à laquelle la négation arithmétique sera appliquée. Si cette chaîne ne peut pas être convertie proprement en une valeur numérique, Perl émettra l'avertissement **Argument "the string" isn't numeric in negation (-) at ...**

L'opérateur unaire "~" effectue la négation bit à bit, c.-à-d. le complément à 1. Par exemple `0666 & ~027` vaut `0640`. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.) Notez que la taille du résultat dépend de la plate-forme : `~0` a une taille de 32 bits sur une plate-forme 32-bit et une taille de 64 bits sur une plate-forme 64-bit. Donc si vous attendez un certain nombre de bits, souvenez-vous d'utiliser l'opérateur `&` pour masquer les bits en excès.

L'opérateur unaire "+" n'a aucun effet même sur les chaînes. Il est pratique pour séparer syntaxiquement un nom de fonction d'une expression entre parenthèses qui autrement aurait été interprétée comme la liste complète des arguments de la fonction. (Voir les exemples de Termes et opérateurs de listes (leftward).)

L'opérateur unaire "\" crée une référence vers ce qui le suit. Voir *perlref*. Ne confondez pas ce comportement avec celui de la barre oblique inversée (backslash) à l'intérieur d'une chaîne bien que les deux formes proposent une sorte de protection contre l'interprétation de ce qui les suit.

### 29.1.7 Opérateurs d'application d'expressions rationnelles

L'opérateur binaire "=" applique un motif de reconnaissance à une expression scalaire. Plusieurs opérations cherchent ou modifient la chaîne \$\_ par défaut. Cet opérateur permet d'appliquer cette sorte d'opérations à d'autres chaînes. L'argument de droite est le motif de recherche, de substitution ou de remplacement. L'argument de gauche est ce qui est supposé être cherché, substitué ou remplacé à la place de la valeur par défaut \$\_. Dans un contexte scalaire, la valeur retournée indique généralement le succès de l'opération. Le comportement dans un contexte de liste dépend de chaque opérateur. Pour plus de détails, voir Opérateurs apostrophe et type apostrophe (§??) et, pour des exemples d'utilisation, voir *perlreftut*.

Si l'argument de droite est une expression plutôt qu'un motif de recherche, de substitution ou de remplacement, il est interprété comme un motif de recherche lors de l'exécution.

L'opérateur binaire "!~" est exactement comme "~" sauf que la valeur retournée est le contraire au sens logique.

### 29.1.8 Opérateurs type multiplication

L'opérateur binaire "\*" multiplie deux nombres.

L'opérateur binaire "/" divise deux nombres.

L'opérateur binaire "%" calcule le modulo de deux nombres. Soit deux opérandes entiers donnés \$a et \$b : si \$b est positif alors \$a % \$b vaut \$a moins le plus grand multiple de \$b qui n'est pas plus grand que \$a. Si \$b est négatif alors \$a % \$b vaut \$a moins le plus petit multiple de \$b qui n'est pas plus petit que \$a (c.-à-d. le résultat est plus petit ou égal à zéro). Remarquez que lorsque vous êtes dans la portée de `use integer`, "%" vous donne accès directement à l'opérateur modulo tel qu'il est défini par votre compilateur C. Cet opérateur n'est pas très bien défini pour des opérandes négatifs mais il s'exécute plus rapidement.

L'opérateur binaire "x" est l'opérateur de répétition. Dans un contexte scalaire, il retourne une chaîne constituée de son opérande de gauche répété le nombre de fois spécifié par son opérande de droite. Dans un contexte de liste, si l'opérande de gauche est entre parenthèses ou est une liste formée par `qw/chaîne/`, il répète la liste. Si l'opérande de droite est nul ou négatif, il retourne un chaîne vide ou une liste vide, selon le contexte.

```
print '-' x 80; # affiche une ligne de '-'

print "\t" x ($tab/8), ' ' x ($tab%8); # tab over

@ones = (1) x 80; # une liste de quatre-vingt 1.
@ones = (5) x @ones; # place tous les éléments à 5.
```

### 29.1.9 Opérateurs type addition

L'opérateur binaire "+" retourne la somme de deux nombres.

L'opérateur binaire "-" retourne la différence de deux nombres.

L'opérateur binaire "." concatène deux chaînes.

### 29.1.10 Opérateurs de décalages

L'opérateur binaire "<<" retourne la valeur de son opérande de gauche décalée vers la gauche d'un nombre de bits spécifié par son opérande de droite. Les arguments devraient être des entiers. (Voir aussi Arithmétique entière.)

L'opérateur binaire ">>" retourne la valeur de son opérande de gauche décalée vers la droite d'un nombre de bits spécifié par son opérande de droite. Les arguments devraient être des entiers. (Voir aussi Arithmétique entière.)

Notez que "<<" et ">>" en Perl sont implémentés en utilisant directement les opérateurs "<<" et ">>" du C. Si `use integer` (voir Arithmétique entière) est actif alors ce sont des entiers signés du C qui sont utilisés et sinon ce sont des entiers non signés. De plus, ces opérations ne génèrent jamais d'entiers plus grands que la limite imposée par le type d'entiers utilisé pour compiler Perl (32 ou 64 bits).

Le résultat obtenu lors d'un dépassement de capacité des entiers n'est pas défini puisqu'il ne l'est pas non plus en C. En d'autres termes, en supposant des entiers sur 32 bits, le résultat de `1 << 32` est indéfini. Un décalage d'un nombre négatif de bits n'est pas défini non plus.

### 29.1.11 Opérateurs unaires nommés

Les différents opérateurs unaire nommés sont traités comme des fonctions à un argument avec des parenthèses optionnelles.

Si un opérateur de liste (`print()`, etc.) ou un opérateur unaire (`chdir()`, etc.) est suivi d'une parenthèse ouvrante, l'opérateur et ses arguments entre parenthèses sont considérés comme de priorité la plus haute exactement comme n'importe quel appel à une fonction. Par exemple, puisque les opérateurs unaires nommés sont plus prioritaires que `||` :

```
chdir $foo || die; # (chdir $foo) || die
chdir($foo) || die; # (chdir $foo) || die
chdir ($foo) || die; # (chdir $foo) || die
chdir +($foo) || die; # (chdir $foo) || die
```

mais puisque `*` est plus prioritaire que les opérateurs unaires nommés :

```
chdir $foo * 20; # chdir ($foo * 20)
chdir($foo) * 20; # (chdir $foo) * 20
chdir ($foo) * 20; # (chdir $foo) * 20
chdir +($foo) * 20; # chdir ($foo * 20)

rand 10 * 20; # rand (10 * 20)
rand(10) * 20; # (rand 10) * 20
rand (10) * 20; # (rand 10) * 20
rand +(10) * 20; # rand (10 * 20)
```

Du point de vue des priorités, les opérateurs de test sur fichiers comme `-f`, `-M`, etc. sont traités comme des opérateurs unaires nommés mais ils ne suivent pas les règles liées aux parenthèses autour des arguments des fonctions. Cela signifie, par exemple, que `-f($file)".bak"` est équivalent à `-f "$file.bak"`.

Voir aussi Termes et opérateurs de listes (leftward) (§??).

### 29.1.12 Opérateurs de comparaisons

L'opérateur binaire `<` renvoie vrai si son opérande gauche est numériquement et strictement plus petit que son opérande droit.

L'opérateur binaire `>` renvoie vrai si son opérande gauche est numériquement et strictement plus grand que son opérande droit.

L'opérateur binaire `<=` renvoie vrai si son opérande gauche est numériquement plus petit ou égal à son opérande droit.

L'opérateur binaire `>=` renvoie vrai si son opérande gauche est numériquement plus grand ou égal à son opérande droit.

L'opérateur binaire `lt` renvoie vrai si son opérande gauche est alphabétiquement et strictement plus petit que son opérande droit.

L'opérateur binaire `gt` renvoie vrai si son opérande gauche est alphabétiquement et strictement plus grand que son opérande droit.

L'opérateur binaire `le` renvoie vrai si son opérande gauche est alphabétiquement plus petit ou égal à son opérande droit.

L'opérateur binaire `ge` renvoie vrai si son opérande gauche est alphabétiquement plus grand ou égal à son opérande droit.

### 29.1.13 Opérateurs d'égalité

L'opérateur binaire `==` renvoie vrai si l'opérande gauche est numériquement égal à l'opérande droit.

L'opérateur binaire `!=` renvoie vrai si l'opérande gauche n'est pas numériquement égal à l'opérande droit.

L'opérateur binaire `<=>` renvoie -1, 0 ou 1 selon que l'opérande gauche est numériquement et respectivement plus petit, égal ou plus grand que l'opérande droit. Si votre plate-forme reconnaît NaN (*not-a-number*, n'est-pas-un-nombre) comme valeur numérique, son utilisation par `<=>` retourne undef. NaN n'est ni plus grand, ni plus petit, ni égal à quoi que ce soit (même pas NaN). `NaN != NaN` retourne vrai comme le fait `NaN !=` n'importe quoi d'autre. Si votre plate-forme ne reconnaît pas NaN alors NaN est simplement une chaîne de caractères qui a 0 pour valeur numérique.

```
perl -le '$a = "NaN"; print "NaN est reconnu" if $a == $a'
perl -le '$a = "NaN"; print "NaN n'est pas reconnu" if $a != $a'
```

L'opérateur binaire "eq" renvoie vrai si l'opérande gauche est égal alphabétiquement à l'opérande droit.

L'opérateur binaire "ne" renvoie vrai si l'opérande gauche n'est pas égal alphabétiquement à l'opérande droit.

L'opérateur binaire "cmp" renvoie -1, 0 ou 1 selon que l'opérande gauche est alphabétiquement et respectivement plus petit, égal ou plus grand que l'opérande droit.

"lt", "le", "ge", "gt" et "cmp" utilisent l'ordre de tri (collation) spécifié par le locale courant si use locale est actif. Voir *perllocale*.

### 29.1.14 Opérateur et bit à bit

L'opérateur binaire "&" renvoie le résultat d'un ET bit à bit entre ses opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

Notez que "&" a une priorité plus faible que les opérateurs de comparaison si bien que les parenthèses sont indispensables dans un test comme ci-dessous :

```
print "Pair\n" if ($x & 1) == 0;
```

### 29.1.15 Opérateurs ou et ou exclusif bit à bit

L'opérateur binaire "|" renvoie le résultat d'un OU bit à bit entre ses deux opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

L'opérateur binaire "^" renvoie le résultat d'un OU EXCLUSIF (XOR) bit à bit entre ses deux opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

Notez que "|" et "^" ont une priorité plus faible que les opérateurs de comparaison si bien que les parenthèses sont indispensables dans un test comme ci-dessous :

```
print "faux\n" if (8 | 2) != 10;
```

### 29.1.16 Et logique style C

L'opérateur binaire "&&" calcule un ET logique rapide. Cela signifie que si l'opérande gauche est faux, l'opérande droit n'est même pas évalué. Le contexte scalaire ou de liste se propage vers l'opérande droit si il est évalué.

### 29.1.17 Défini-ou logique style C

Bien que n'ayant pas d'équivalent direct en C, l'opérateur Perl // s'apparente à un ou style C. En fait, c'est exactement la même chose que || sauf qu'il ne teste pas la véracité de son opérande gauche mais le fait qu'il soit défini. Donc, \$a // \$b est similaire à defined(\$a) || \$b (sauf qu'il retourne la valeur de \$a au lieu de la valeur de defined(\$a)) et est exactement équivalent à defined(\$a) ? \$a : \$b. C'est très pratique pour donner des valeurs par défaut à des variables. On peut aussi tester si au moins l'une des variables \$a ou \$b est définie en écrivant defined(\$a // \$b).

### 29.1.18 Ou logique style C

L'opérateur binaire "||" calcule un OU logique rapide. Cela signifie que si l'opérande gauche est vrai, l'opérande droit n'est même pas évalué. Le contexte scalaire ou de liste se propage vers l'opérande droit si il est évalué.

Les opérateurs ||, // et && renvoient la dernière valeur évaluée (contrairement aux homologues en C, || et && qui renvoient 0 ou 1). Donc, un moyen raisonnablement portable de trouver le répertoire home peut être :

```
$home = $ENV{HOME} // $ENV{LOGDIR} //
(getpwuid($<))[7] // die "You're homeless!\n";
```

En particulier, cela signifie que vous ne devriez pas les utiliser pour choisir entre deux agrégats dans une <affectation :>

```
@a = @b || @c; # c'est pas bon
@a = scalar(@b) || @c; # voila ce que ça signifie
@a = @b ? @b : @c; # cela marche très bien par contre
```

Pour remplacer d'une manière plus lisible l'usage de `&&`, `//` et `||` pour contrôler un flot d'opérations, Perl propose les opérateurs `and`, `err` et `or` (voir plus bas). Le comportement d'évaluation rapide est le même. En revanche, comme la priorité de `"and"`, `"err"` et `"or"` est plus faible, vous pouvez les utiliser après les opérateurs de listes sans ajouter de parenthèses :

```
unlink "alpha", "beta", "gamma"
 or gripe(), next LINE;
```

Avec les opérateurs à la C, vous auriez dû l'écrire :

```
unlink("alpha", "beta", "gamma")
 || (gripe(), next LINE);
```

En revanche, l'utilisation de `"or"` lors d'une affectation ne donne pas ce que vous voulez; voir plus bas.

### 29.1.19 Opérateurs d'intervalle

L'opérateur binaire `".."` est l'opérateur d'intervalle qui est en fait deux opérateurs totalement différents selon le contexte. Dans un contexte de liste, il renvoie une liste de valeurs commençant à la valeur de son opérande gauche et se terminant à la valeur de son opérande droit (par pas de 1). Si la valeur de gauche est plus petite que la valeur de droite, il retourne une liste vide. C'est pratique pour écrire des boucles `foreach` (`1..10`) et pour des opérations de remplacement sur des tableaux. Dans l'implémentation actuelle, aucun tableau temporaire n'est généré lorsque l'opérateur d'intervalle est utilisé comme expression de boucles `foreach` mais les vieilles versions de Perl peuvent consommer énormément de mémoire lorsque vous écrivez quelque chose comme :

```
for (1 .. 1_000_000) {
 # code
}
```

L'opérateur d'intervalle fonctionne aussi avec des chaînes de caractères, en utilisant l'auto-incrémentation (voir ci-dessous).

Dans un contexte scalaire, `".."` renvoie une valeur booléenne. L'opérateur est bi-stable comme un interrupteur et simule l'opérateur d'intervalle de ligne (virgule) de `sed`, de `awk` et d'autres éditeurs. Chaque opérateur `".."` conserve en mémoire son propre état booléen. Il reste faux tant que son opérande gauche est faux. Puis dès que son opérande gauche devient vrai, il reste vrai jusqu'à ce que son opérande droit soit vrai. *APRÈS* quoi, l'opérateur d'intervalle redevient faux. Il ne redevient faux que le prochaine fois que l'opérateur d'intervalle est évalué. Il peut tester l'opérande droit et devenir faux lors de la même évaluation où il devient vrai (comme dans `awk`) mais il retournera encore une fois vrai. Si vous ne voulez pas qu'il teste l'opérande droit avant la prochaine évaluation (comme dans `sed`), utilisez trois points (`"..."`) à la place de deux. Pour tout le reste, `"..."` se comporte exactement comme `".."`.

L'opérande droit n'est pas évalué tant que l'opérateur est dans l'état "faux" et l'opérande gauche n'est pas évalué tant que l'opérateur est dans l'état "vrai". La priorité de l'opérateur intervalle est un peu plus basse que celle de `||` et `&&`. La valeur retournée est soit la chaîne vide pour signifier "faux" soit un numéro de séquence (commençant à 1) pour "vrai". Le dernier numéro de séquence d'un intervalle est suivi de la chaîne "E0" ce qui ne perturbe pas sa valeur numérique mais vous donne quelque chose à chercher si vous voulez exclure cette dernière valeur. Vous pouvez exclure la première valeur en demandant une valeur supérieure à 1.

Si l'un des opérandes de l'opérateur intervalle pris dans un contexte scalaire est une expression constante, alors cet opérande est considéré comme vrai si il est égal (`==`) au numéro de ligne courant (la variable `$.`).

Plus précisément, la comparaison réellement effectuée est `int(EXPR) == int(EXPR)` ce qui importe uniquement si vous utilisez une expression non entière. Lorsque `$.` est utilisé implicitement, comme dans expliqué dans la paragraphe précédent, la véritable comparaison utilisée est `int(EXPR) == int($.)`. Plus encore, `"span" .. "spat"` ou `2.18 .. 3.14` dans un contexte scalaire ne font pas ce que vous voulez puisque chacun de leurs opérandes est évalué en utilisant sa représentation entière.

Exemples :

Comme opérateur scalaire :

```

if (101 .. 200) { print; } # affiche la seconde centaine de lignes
 # raccourci pour
 # if ($. == 101 .. $. == 200) ...

next line if (1 .. /^$/); # saut des lignes d'en-têtes
 # raccourci pour
 # ... if ($. == 1 .. /^$/);

s/^/> / if (/^$/ .. eof()); # place le corps comme citation

analyse d'e-mail
while (<>) {
 $in_header = 1 .. /^$/;
 $in_body = /^$/ .. eof();
 if ($in_header) {
 # ...
 } else { # in body
 # ...
 }
} continue {
 close ARGV if eof; # réinitialisation de $. à chaque fichier
}

```

Voici un exemple illustrant les différences entre les deux opérateurs d'intervalle :

```

@lines = (" - Foo",
 "01 - Bar",
 "1 - Baz",
 " - Quux");

foreach (@lines) {
 if (/0/ .. /1/) {
 print "$_\n";
 }
}

```

Ce programme n'affiche que la ligne contenant "Bar". En utilisant l'opérateur d'intervalle `...`, il afficherait aussi la ligne contenant "Baz".

Et maintenant quelques exemple en tant qu'opérateur de liste :

```

for (101 .. 200) { print; } # affiche $_ 100 fois
@foo = @foo[0 .. $#foo]; # un no-op coûteux
@foo = @foo[$#foo-4 .. $#foo]; # extraction des 5 derniers items

```

L'opérateur intervalle (dans un contexte de liste) utilise l'algorithme magique d'auto-incrémentation si les opérandes sont des chaînes. Vous pouvez écrire :

```
@alphabet = ('A' .. 'Z');
```

pour obtenir toutes les lettres de l'alphabet anglais ou

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

pour obtenir les chiffres hexadécimaux ou

```
@z2 = ('01' .. '31'); print $z2[$mjour];
```

pour obtenir des dates avec des zéros. Si la valeur finale donnée n'est pas dans la séquence produite par l'incrémentement magique, la séquence continue jusqu'à ce que la prochaine valeur soit plus longue que la valeur finale spécifiée.

Comme chaque opérande est évalué sous forme entière, `2.18 .. 3.14` retournera deux éléments dans un contexte de liste.

```
@list = (2.18 .. 3.14); # identique à @list = (2 .. 3);
```



### 29.1.20 Opérateur conditionnel

L'opérateur "?" est l'opérateur conditionnel exactement comme en C. Il travaille presque comme un si-alors-sinon. Si l'argument avant le ? est vrai, l'argument avant le : est renvoyé sinon l'argument après le : est renvoyé. Par exemple :

```
printf "J'ai %d chien%s.\n", $n,
 ($n == 1) ? '' : "s";
```

Le contexte scalaire ou de liste se propage au second ou au troisième argument quelque soit le choix.

```
$a = $ok ? $b : $c; # un scalaire
@a = $ok ? @b : @c; # un tableau
$a = $ok ? @b : @c; # oups, juste un compte !
```

On peut affecter quelque chose à l'opérateur si le second ET le troisième arguments sont des lvalues légales (ce qui signifie qu'on peut leur affecter quelque chose) :

```
($a_or_b ? $a : $b) = $c;
```

Il n'y a aucune garantie que cela contribue à la lisibilité de votre programme.

Puisque l'opérateur produit un résultat affectable, l'usage d'affectation sans parenthèse peut amener quelques problèmes. Par exemple, la ligne suivante :

```
$a % 2 ? $a += 10 : $a += 2
```

signifie réellement :

```
(($a % 2) ? ($a += 10) : $a) += 2
```

au lieu de :

```
($a % 2) ? ($a += 10) : ($a += 2)
```

Cela aurait probablement pu être écrit plus simplement :

```
$a += ($a % 2) ? 10 : 2;
```

### 29.1.21 Opérateurs d'affectation

"=" est l'opérateur habituel d'affectation.

Les opérateurs d'affectation fonctionnent comme en C. Donc :

```
$a += 2;
```

est équivalent à :

```
$a = $a + 2;
```

quoique sans dupliquer les éventuels effets de bord que le déréférencement de la lvalue pourraient déclencher, comme par exemple avec tie(). Les autres opérateurs d'affectation fonctionnent de la même manière. Voici ceux qui sont reconnus :

```
**= += *= &= <<= &&=
 -= /= |= >>= ||=
 .= %= ^=
 x=
```

Remarque: bien que regroupés par famille, tous ces opérateurs ont la même priorité que l'affectation.

Au contraire du C, les opérateurs d'affectation produisent une lvalue valide. Modifier une affectation est équivalent à faire l'affectation puis à modifier la variable qui vient d'être affectée. C'est très pratique pour modifier une copie de quelque chose comme dans :

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

De même :

```
($a += 2) *= 3;
```

est équivalent à :

```
$a += 2;
$a *= 3;
```

De manière similaire, une affectation vers une liste dans un contexte de liste produit la liste des lvalues affectées et dans un contexte scalaire produit le nombre d'éléments présents dans l'opérande de droite de l'affectation.

### 29.1.22 Opérateur virgule

L'opérateur binaire "," est l'opérateur virgule. Dans un contexte scalaire, il évalue son opérande gauche et jette le résultat puis il évalue son opérande droit et retourne cette valeur. C'est exactement comme l'opérateur virgule du C.

Dans un contexte de liste, c'est tout simplement le séparateur d'arguments de la liste. Il insère ses deux opérandes dans la liste.

L'opérateur => est un synonyme de la virgule sauf qu'il contraint un mot (constitué uniquement de caractères de mots) à sa gauche à être interprété comme une chaîne (depuis la version 5.001). Cela inclut les mots qui, dans un autre contexte, auraient pu être considérés comme des constantes ou des appels de fonctions.

```
use constant FOO => "quelque chose";
```

```
my %h = (FOO => 23);
```

qui est équivalent à :

```
my %h = ("FOO", 23);
```

et *non* à :

```
my %h = ("quelque chose", 23);
```

Si son argument de gauche n'est pas un mot, il est tout d'abord interprété comme une expression puis, ensuite, c'est sa valeur en tant que chaîne de caractères qui est utilisée.

### 29.1.23 Opérateurs de listes (Rightward)

Du côté droit d'un opérateur de liste, il a une priorité très basse qui permet de maîtriser toutes les expressions présentes séparées par des virgules. Les seuls opérateurs de priorité inférieure sont les opérateurs logiques "and", "or" et "not" qui peuvent être utiliser pour évaluer plusieurs appels à des opérateurs de liste sans l'ajout de parenthèses supplémentaires :

```
open HANDLE, "filename"
 or die "Can't open: $!\n";
```

Voir aussi la discussion sur les opérateurs de liste dans Termes et opérateurs de liste (leftward).

### 29.1.24 Non (not) logique

L'opérateur unaire "not" renvoie la négation logique de l'expression à sa droite. Il est équivalent à "!" sauf sa priorité beaucoup plus basse.

### 29.1.25 Et (and) logique

L'opérateur binaire "and" renvoie la conjonction logique des deux expressions qui l'entourent. Il est équivalent à "&&" sauf sa priorité beaucoup plus basse. Cela signifie qu'il est rapide: l'expression de droite est évaluée uniquement si celle de gauche est vraie.

### 29.1.26 Ou (or), ou exclusif (xor) et défini-ou (err) logiques

L'opérateur binaire "or" renvoie la disjonction logique des deux expressions qui l'entourent. Il est équivalent à "||" sauf sa priorité beaucoup plus basse. C'est pratique pour contrôler une suite d'opérations :

```
print FH $data or die "Can't write to FH: $!";
```

Cela signifie qu'il est rapide: l'expression de droite est évaluée uniquement si celle de gauche est fausse. À cause de sa priorité, vous devriez l'éviter dans les affectations et ne l'utiliser que pour du contrôle d'opérations.

```
$a = $b or $c; # bug: c'est pas bon
($a = $b) or $c; # voilà ce que ça signifie
$a = $b || $c; # il vaut mieux l'écrire ainsi
```

Au contraire, lorsque l'affectation est dans un contexte de liste et que vous voulez utiliser "||" pour du contrôle, il vaut mieux utiliser "or" pour que l'affectation soit prioritaire.

```
@info = stat($file) || die; # holà, sens scalaire de stat !
@info = stat($file) or die; # meilleur, @info reçoit ce qu'il faut
```

Bien sûr, il est toujours possible d'utiliser les parenthèses.

L'opérateur binaire "err" est l'équivalent de "//" (c'est comme un "ou" binaire sauf qu'il teste si son opérande gauche est défini au lieu de tester si il est vrai). Il y a deux moyens de mémoriser "err" : soit parce que de nombreuses fonctions retournent undef en cas d'erreur, soit comme une sorte de correction (`$a=( $b err 'défaut' )`).

L'opérateur binaire "xor" renvoie le ou exclusif des deux expressions qui l'entourent. Il ne peut évidemment pas être rapide.

### 29.1.27 Opérateurs C manquant en Perl

Voici ce qui existe en C et que Perl n'a pas :

#### & unaire

Opérateur adresse-de. (Voir l'opérateur "\" pour prendre une référence.)

#### \* unaire

Opérateur contenu-de ou de déréférencement. (Les opérateurs de déréférencement de Perl sont des préfixes typés : \$, @, % et &.)

#### (TYPE)

Opérateur de conversion de type (de cast).

### 29.1.28 Opérateurs apostrophe et apparentés

Bien qu'habituellement nous pensions aux apostrophes (et autres guillemets) pour des valeurs littérales, en Perl, elles fonctionnent comme des opérateurs et proposent différents types d'interpolation et de capacités de reconnaissance de motif. Perl fournit des caractères standard pour cela mais fournit aussi le moyen de choisir vos propres caractères. Dans la table suivante, le {} représente n'importe quelle paire de délimiteurs que vous aurez choisie.

Standard	Générique	Signification	Interpolation
'	q{}	Littérale	non
""	qq{}	Littérale	oui
' '	qx{}	Commande	oui*
	qw{}	Liste de mots	non
//	m{}	Reconnaissance de motif	oui*
	qr{}	Motif	oui*
	s{}	Substitution	oui*
	tr{}	Translittération	non (mais voir plus bas)
<<EOF		here-doc	oui*

\* sauf si les délimiteurs sont de simples apostrophes (').

Les délimiteurs qui ne marchent pas par deux utilisent le même caractère au début et à la fin par contre les quatre sortes de parenthèses (parenthèses, crochets, accolades et inférieur/supérieur) marchent par deux et peuvent être imbriquées ce qui signifie que :

```
q{foo{bar}baz}
```

est la même chose que :

```
'foo{bar}baz'
```

Remarquez par contre que cela ne fonctionne pas toujours. Par exemple :

```
$s = q{ if($a eq "") ... }; # Mauvais
```

est une erreur de syntaxe. Le module `Text::Balanced` (disponible sur CPAN et intégré à la distribution Perl depuis la version 5.8) est capable de gérer cela correctement.

Il peut y avoir des espaces entre l'opérateur et le caractère délimiteur sauf lorsque # est utilisé. `q#foo#` est interprété comme la chaîne `foo` alors que `q #foo#` est l'opérateur `q` suivi d'un commentaire. Ses arguments sont alors pris sur la ligne suivante. Cela permet d'écrire :

```
s {foo} # Remplace foo
 {bar} # par bar.
```

Les séquences d'échappement suivantes sont disponibles dans les constructions qui font de l'interpolation et dans les translittérations :

<code>\t</code>	tabulation	(HT, TAB)
<code>\n</code>	nouvelle ligne	(LF, NL)
<code>\r</code>	retour chariot	(CR)
<code>\f</code>	page suivante	(FF)
<code>\a</code>	alarme (bip)	(BEL)
<code>\e</code>	escape	(ESC)
<code>\033</code>	caractère en octal	(ESC)
<code>\x1B</code>	caractère hexadécimal	(ESC)
<code>\x{236a}</code>	caractère hexadécimal long (SMILEY)	
<code>\c[</code>	caractère de contrôle	(ESC)
<code>\N{nom}</code>	caractère Unicode nommé	

**Note** : la séquence `\v` (pour une tabulation verticale - VT - ASCII 11) qui existe en C et dans d'autres langages n'est pas reconnue en Perl.

Les séquences d'échappement suivantes sont disponibles dans les constructions qui font de l'interpolation mais pas dans les translittérations :

```

\l convertit en minuscule le caractère suivant
\u convertit en majuscule le caractère suivant
\L convertit en minuscule jusqu'au prochain \E
\U convertit en majuscule jusqu'au prochain \E
\E fin de modification de casse
\Q désactive les méta-caractères de motif jusqu'au prochain \E

```

Si `use locale` est actif, la table de conversion majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est celle du locale courant. Voir *perllocale*. En Unicode (par exemple avec `\N{}` ou avec des caractères larges codés en hexadécimal avec une valeur supérieure à 0x100), la table de conversion majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est définie par les conventions Unicode. Pour la documentation de `\N{nom}`, voir *charnings*.

Tous les systèmes utilisent le `"\n"` virtuel pour représenter une terminaison de ligne appelée "newline" ou "nouvelle ligne". Il n'existe pas de caractère physique invariant pour représenter ce caractère "newline". C'est une illusion qu'essayent conjointement de maintenir le système d'exploitation, les pilotes de périphériques, la bibliothèque C et Perl. Tous les systèmes ne lisent pas `"\r"` comme le CR ASCII ni `"\n"` comme le LF ASCII. Par exemple, sur Mac, ils sont inversés et sur des systèmes sans terminaison de ligne, écrire `"\n"` peut ne produire aucun donnée. En général, utilisez `"\n"` lorsque vous pensez "newline" pour votre système mais utilisez le littéral ASCII quand voulez un caractère exact. Par exemple, de nombreux protocoles réseaux attendent et préfèrent un CR+LF (`"\015\012"` ou `"\cJ\cM"`) comme terminaison de ligne. La plupart acceptent un simple `"\012"` et ne tolèrent que très rarement un simple `"\015"`. Si vous prenez l'habitude d'utiliser `"\n"` sur les réseaux, vous aurez des problèmes un jour.

Dans les constructions qui font appel à de l'interpolation, les variables commençant par `"$"` ou `"@"` sont interpolées. Les variables utilisant des indices comme dans `$a[3]` ou `$href->{key}[0]` sont aussi interpolées comme le sont les tranches de tableaux ou de tables de hachage. Par contre, les appels de méthodes comme `$obj->meth` ne le sont pas.

Un tableau (ou une tranche de tableau) est interpolé par la suite de valeurs dans l'ordre séparées par la valeur de `"$"`, ce qui est équivalent à `join "$", @tableau`. Les tableaux dont le nom utilisent des caractères spéciaux comme `@+` ne sont interpolés que si on entoure le nom par des accolades `@{+}`.

Vous ne pouvez pas inclure littéralement les caractères `$` et `@` à l'intérieur d'une séquence `\Q`. Tels quels, ils se référerait à la variable correspondante. Précédés d'un `\`, ils correspondraient à la chaîne `\$` ou `\@`. Vous êtes obligés d'écrire quelque chose comme `m/\Quser\E@\Qhost/`.

Les motifs sont sujets à un niveau supplémentaire d'interprétation en tant qu'expression rationnelle. C'est fait lors d'une seconde passe après l'interpolation des variables si bien qu'une expression rationnelle peut-être introduite dans un motif via une variable. Si ce n'est pas ce que vous voulez, utilisez `\Q` pour interpoler une variable littéralement.

Mis à part ce qui précède, il n'y pas de multiples niveaux d'interpolation. En particulier et contrairement à ce qu'attendaient des programmeurs shell, les accents graves (ou backticks) *NE* sont *PAS* interpolées à l'intérieur des guillemets et les apostrophes n'empêchent pas l'évaluation des variables à l'intérieur des guillemets.

## 29.1.29 Opérateurs d'expression rationnelle

Nous discuterons ici des opérateurs style apostrophe qui implique une reconnaissance de motif ou une action s'y rapportant.

### ?MOTIF?

C'est la même chose qu'une recherche par `/motif/` sauf qu'elle n'est reconnue qu'une seule fois entre chaque appel à l'opérateur `reset()`. C'est une optimisation pratique si vous ne recherchez que la première occurrence de quelque chose dans chaque fichier ou groupes de fichiers par exemple. Seuls les motifs `??` locaux au package courant sont réinitialisés par `reset()`.

```

while (<>) {
 if (??) {
 # ligne blanche entre en-tête et corps
 }
} continue {
 reset if eof; # réinitialisation de ?? pour le fichier suivant
}

```

Cet usage est plus ou moins désapprouvé et pourrait être supprimé dans une version future de Perl. Peut-être vers l'année 2168.

### m/MOTIF/cgimosx

#### /MOTIF/cgimosx

Effectue la recherche d'un motif d'expression rationnelle dans une chaîne et, dans un contexte scalaire, renvoie vrai en cas de succès ou faux sinon. Si aucune chaîne n'est spécifiée via l'opérateur =~ ou l'opérateur !~, c'est dans la chaîne \$\_ que s'effectue la recherche. (La chaîne spécifiée par =~ n'est pas nécessairement une lvalue – cela peut être le résultat de l'évaluation d'une expression.) Voir aussi *perlre*. Voir *perllocale* pour des informations supplémentaires qui s'appliquent lors de l'usage de `use locale`.

Les options (ou modificateurs) sont :

- c Ne pas réinitialiser la position de recherche lors d'un échec avec /g.
- g Recherche globale, c.-à-d. trouver toutes les occurrences.
- i Reconnaissance de motif indépendamment de la casse (majuscules/minuscules).
- m Traitement de chaîne comme étant multi-lignes.
- o Compilation du motif uniquement la première fois.
- s Traitement de la chaîne comme étant une seule ligne.
- x Utilisation des expressions rationnelles étendues.

Si "/" est le délimiteur alors le m initial est optionnel. Avec le m vous pouvez utiliser n'importe quelle paire de caractères ni alphanumériques ni blancs comme délimiteur. C'est particulièrement pratique pour les chemins d'accès Unix qui contiennent des "/" afin d'éviter le LTS (leaning toothpick syndrome). Si "?" est le délimiteur alors la règle "ne-marche-qu'une-fois" de ?MOTIF? s'applique. Si "" est le délimiteur alors aucune interpolation n'est effectuée sur le MOTIF.

MOTIF peut contenir des variables qui seront interpolées (et le motif recompilé) chaque fois que la recherche du motif est effectuée sauf si le délimiteur choisi est une apostrophe. (Remarquez que \$(, \$) et \$| ne peuvent pas être interpolés puisqu'ils ressemblent à des tests de fin de chaîne.) Si vous voulez qu'un motif ne soit compilé qu'une seule fois, ajoutez /o après le dernier délimiteur. Ceci évite des coûteuses recompilations lors de l'exécution et c'est utile lorsque la valeur interpolée ne doit pas changer pendant la durée de vie du script. Par contre, ajouter /o suppose que vous ne changerez pas la valeur des variables présentes dans le motif. Si vous les changez, Perl ne s'en apercevra même pas. Voir aussi qr/CHAINE/imosx (§??).

Si l'évaluation du MOTIF est la chaîne vide, la dernière expression rationnelle *reconnue* est utilisée à la place. Dans ce cas, seuls les modificateurs g et c du motif vide sont pris en compte - les autres modificateurs seront ceux de l'expression rationnelle utilisée. Si aucun motif n'a été précédemment reconnu, le motif utilisé (silencieusement) sera le motif vide (qui est toujours reconnaissable).

Notez qu'il est possible que Perl confonde // (l'expression rationnelle vide) et // (l'opérateur défini-ou). Habituellement, Perl se débrouille assez bien pour cela mais quelques cas pathologiques peuvent se présenter, comme, par exemple, \$a/// (est-ce (\$a) / (//) ou \$a // /?) ou `print $fh //` (est-ce `print $fh(//` ou `print($fh // ?)`). Dans tous ces exemples, Perl choisira l'opérateur défini-ou. Pour choisir l'expression rationnelle vide, il vous suffit d'ajouter des parenthèses ou un espace pour lever l'ambiguïté ou encore préfixer votre expression rationnelle vide par un m (// devient donc m//).

Si l'option /g n'est pas utilisée, dans un contexte de liste, m// retourne une liste constituée de tous les sous-motifs reconnus par des parenthèses dans le motif (c.-à-d. \$1, \$2, \$3...). (Remarquez que \$1, \$2, etc. sont aussi mises à jours ce qui diffère du comportement de Perl 4.) Lorsqu'il n'y a pas de parenthèses dans le motif, la valeur retournée est la liste (1) en cas de succès. Qu'il y ait ou non de parenthèses, une liste vide est retournée en cas d'échec.

Exemples :

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo(); # appel de foo si désiré

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

le grep du pauvre
$arg = shift;
while (<>) {
 print if /$arg/o; # compile une seule fois
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

Ce dernier exemple découpe \$foo en trois parties (le deux premiers mots et le reste de la ligne) et les affecte à \$F1, \$F2 et \$Etc. La condition est vraie si au moins une des variables est affectée, c.-à-d. si le motif est reconnu.

Le modificateur /g spécifie une recherche globale du motif – c'est à dire la recherche d'autant de correspondances que possible dans la chaîne. Le comportement dépend du contexte. Dans un contexte de liste, c'est la liste de toutes les sous-chaînes reconnues par les sous-motifs (entre parenthèses) du motif qui est retournée. En l'absence de parenthèses, c'est la liste de toutes les chaînes correspondant au motif qui est retournée, comme si il y avait des parenthèses autour du motif lui-même.

Dans un contexte scalaire, chaque exécution de m//g trouve la prochaine correspondance et retourne vrai si il y a correspondance et faux si il n'y en a plus. La position après la dernière correspondance peut-être lue et modifiée par la fonction pos(); voir pos() in *perlfunc*. Normalement, un échec de la recherche réinitialise la position de recherche au début de la chaîne sauf si vous ajoutez le modificateur /c (par ex. m//gc). La modification de la chaîne sur laquelle à lieu la recherche réinitialise aussi la position de recherche.

Vous pouvez mélanger des reconnaissances m//g avec des m/\G.../g où \G est l'assertion de longueur nulle qui est reconnue à la position exacte où, si elle existe, s'est arrêtée la précédente recherche m//g. Sans le modificateur /g, l'assertion pourra encore s'ancrer sur pos(), mais cette reconnaissance n'est évidemment essayée qu'une seule fois. L'application de \G sans /g sur une chaîne de caractères sur laquelle aucune reconnaissance utilisant /g n'a encore été faite, revient à utiliser l'assertion \A pour reconnaître le début de la chaîne. Notez aussi que, actuellement, \G ne fonctionne bien que si il est utilisé au tout début du motif.

Exemples :

```
contexte de liste
($one,$five,$fifteen) = ('uptime' =~ /(\d+\.\d+)/g);

contexte scalaire
$/ = "";
while (defined($paragraph = <>)) {
 while ($paragraph =~ /[a-z](['"])*[.!?]+(['"])*\s/g) {
 $sentences++;
 }
}
print "$sentences\n";

utilisation de m//gc avec \G
$_ = "ppooqppq";
while ($i++ < 2) {
 print "1: ";
 print $1 while /(o)/gc; print ", pos=", pos, "\n";
 print "2: ";
 print $1 if /\G(q)/gc; print ", pos=", pos, "\n";
 print "3: ";
 print $1 while /(p)/gc; print ", pos=", pos, "\n";
}
print "Final: '$1', pos=",pos,"\n" if /\G(.)/;
```

Le dernier exemple devrait afficher :

```
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
Final: 'q', pos=8
```

Remarquez que la reconnaissance finale reconnaît q au lieu de p, comme l'aurait fait une reconnaissance sans \G. Notez aussi que cette reconnaissance finale ne met pas à jour pos (pos n'est mis à jour qu'avec /g). Si vous obtenez un p lors de cette reconnaissance finale, cela indique que vous utilisez une vieille version de Perl (avant 5.6.0).

Une construction idiomatique pratique pour des analyseurs à la lex est /\G.../gc. Vous pouvez combiner plusieurs expressions rationnelles de ce type pour traiter une chaîne partie par partie en faisant différentes actions selon l'expression qui est reconnue. Chaque expression essaye de correspondre là où la précédente s'est arrêtée.

```
$_ = <<'EOL';
$url = new URI::URL "http://www/"; die if $url eq "xXx";
```

```

EOL
LOOP:
{
 print(" chiffres"), redo LOOP if /\G\d+\b[,;]?\s*/gc;
 print(" minuscule"), redo LOOP if /\G[a-z]+\b[,;]?\s*/gc;
 print(" MAJUSCULE"), redo LOOP if /\G[A-Z]+\b[,;]?\s*/gc;
 print(" Capitalisé"), redo LOOP if /\G[A-Z][a-z]+\b[,;]?\s*/gc;
 print(" MiXtE"), redo LOOP if /\G[A-Za-z]+\b[,;]?\s*/gc;
 print(" alphanumérique"), redo LOOP if /\G[A-Za-z0-9]+\b[,;]?\s*/gc;
 print(" autres"), redo LOOP if /\G[^A-Za-z0-9]+/gc;
 print ". C'est tout !\n";
}

```

Voici la sortie (découpée en plusieurs lignes) :

```

autres minuscule autres minuscule MAJUSCULE autres
MAJUSCULE autres minuscule autres minuscule autres
minuscule minuscule autres minuscule minuscule autres
MiXtE autres. C'est tout !

```

### q/CHAINE/

#### 'CHAINE'

Une apostrophe pour une chaîne littérale. Un backslash (une barre oblique inverse) représente un backslash sauf si il est suivi par le délimiteur ou par un autre backslash auquel cas le délimiteur ou le backslash est interpolé.

```

$foo = q!I said, "You said, 'She said it.'";
$bar = q('This is it. ');
$baz = '\n'; # une chaîne de deux caractères

```

### qq/CHAINE/

#### "CHAINE"

Des guillemets pour une chaîne interpolée.

```

$_ .= qq
 (** La ligne précédente contient le gros mot "$1".\n)
 if /\b(tcl|java|python)\b/i; # :-)
$baz = "\n"; # une chaîne d'un caractère

```

### qr/CHAINE/imosx

Cette opérateur considère *CHAINE* comme une expression rationnelle (et la compile éventuellement). *CHAINE* est interpolée de la même manière que *MOTIF* dans *m/MOTIF/*. Si "" est utilisé comme délimiteur, aucune interpolation de variables n'est effectuée. Renvoie une expression Perl qui peut être utilisée à la place de l'expression */CHAINE/imosx*.

Par exemple :

```

$rex = qr/ma.CHAINE/is;
s/$rex/foo/;

```

est équivalent à :

```

s/ma.CHAINE/foo/is;

```

Le résultat peut être utilisé comme motif dans une recherche de correspondance :

```

$re = qr/$motif/;
$string =~ /foo{$re}bar/; # peut être interpolée dans d'autres motifs
$string =~ $re; # ou utilisée seule
$string =~ /$re/; # ou de cette manière

```

Puisque Perl peut compiler le motif lors de l'exécution de l'opérateur *qr()*, l'utilisation de *qr()* peut augmenter les performances dans quelques situations, notamment si le résultat de *qr()* est utilisé seul :

```

sub match {
 my $patterns = shift;
 my @compiled = map qr/$_/i, @$patterns;
 grep {

```



```

 my $success = 0;
 foreach my $pat @compiled {
 $success = 1, last if /$pat/;
 }
 $success;
} @_;
}

```

La précompilation du motif en une représentation interne lors du `qr()` évite une recompilation à chaque fois qu'une recherche `/$pat/` est tentée. (Notez que Perl a de nombreuses autres optimisations internes mais aucune n'est déclenchée dans l'exemple précédent si nous n'utilisons pas l'opérateur `qr()`.)

Les options sont :

```

i Motif indépendant de la casse (majuscules/minuscules).
m Traitement de chaîne comme étant multi-lignes.
o Compilation du motif uniquement la première fois.
s Traitement de la chaîne comme étant une seule ligne.
x Utilisation des expressions rationnelles étendues.

```

Voir *perlre* pour de plus amples informations sur la syntaxe correcte de CHAINE et pour une description détaillée de la sémantique des expressions rationnelles.

### **qx/CHAINE/**

#### **'CHAINE'**

Une chaîne qui est (éventuellement) interpolée puis exécutée comme une commande système par `/bin/sh` ou équivalent. Les jokers, tubes (pipes) et redirections sont pris en compte. L'ensemble de la sortie standard de la commande est renvoyé; la sortie d'erreur n'est pas affectée. Dans un contexte scalaire, le résultat est retourné comme une seule chaîne (potentiellement multi-lignes) ou `undef` si la commande échoue. Dans un contexte de liste, le résultat est une liste de lignes (selon la définition des lignes donnée par `$/` ou `$INPUT_RECORD_SEPARATOR`) ou la liste vide si la commande échoue.

Puisque les apostrophes inverses (backticks) n'affectent pas la sortie d'erreur, il vous faut utiliser la (les) syntaxe(s) de redirection du shell (en supposant qu'elle(s) existe(nt)) afin de capter les erreurs. Pour récupérer les sorties `STDOUT` et `STDERR` d'une commande :

```
$output = `cmd 2>&1`;
```

Pour récupérer `STDOUT` et faire disparaître `STDERR` :

```
$output = `cmd 2>/dev/null`;
```

Pour récupérer `STDERR` et faire disparaître `STDOUT` (l'ordre est ici très important) :

```
$output = `cmd 2>&1 1>/dev/null`;
```

Pour échanger `STDOUT` et `STDERR` afin de récupérer `STDERR` tout en laissant `STDOUT` s'afficher normalement :

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

Pour récupérer `STDOUT` et `STDERR` séparément, il est plus simple de les rediriger séparément vers des fichiers qui seront lus lorsque l'exécution de la commande sera terminée :

```
system("program args 1>program.stdout 2>program.stderr");
```

L'utilisation de l'apostrophe comme délimiteur protège la commande de l'interpolation normalement effectuée par Perl sur les chaînes entre guillemets. La chaîne est passée tel quelle au shell :

```

$perl_info = qx(ps $$); # Le $$ de Perl
$shell_info = qx'ps $$'; # Le $$ du nouveau shell

```

Remarquez que la manière dont la chaîne est évaluée est entièrement dépendante de l'interpréteur de commandes de votre système. Sur la plupart des plates-formes, vous aurez à protéger les méta-caractères du shell si vous voulez qu'ils soient traités littéralement. En pratique, c'est difficile à faire, surtout qu'il n'est pas toujours facile de savoir quels caractères doivent être protégés. Voir *perlsec* pour un exemple propre et sûr d'utilisation manuelle de `fork()` et `exec()` pour émuler proprement l'utilisation des backticks.

Sur certaines plates-formes (particulièrement celles style DOS) le shell peut ne pas être capable de gérer des commandes multi-lignes. Dans ce cas, l'usage de passages à la ligne dans la chaîne de commande ne donne pas ce que vous voulez. Vous pouvez évaluer plusieurs commandes sur une seule et même ligne et les séparant par le caractère

de séparation de commandes si votre shell le supporte (par ex. ; pour la plupart des shells Unix; & sur le cmd shell de Windows NT).

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant de lancer la commande mais cela n'est pas supporté sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionner la variable \$| (\$AUTOFLUSH en anglais) ou appelé la méthode autoflush() des objets IO: :Handle pour chacun des descripteurs ouverts.

N'oubliez pas que certains shells ont quelques restrictions sur la longueur de la ligne de commande. Vous devez vous assurer que votre chaîne n'excède pas cette limite même après interpolation. Voir les notes spécifiques à votre plate-forme pour plus de détails sur votre environnement particulier.

L'utilisation de cet opérateur peut aboutir à des programmes difficiles à porter puisque les commandes shells appelées varient d'un système à l'autre et peuvent parfois ne pas exister du tout. À titre d'exemple, la commande type du shell POSIX est très différente de la commande type sous DOS. Cela ne signifie pas qu'il vous faut à tout prix éviter cet opérateur lorsque c'est le bon moyen de faire ce que vous voulez. Perl a été conçu pour être un "glue language"... La seule chose qui compte c'est que vous sachiez ce que vous faites.

Voir Opérateurs d'E/S (§??) pour des plus amples informations.

### qw/CHAINE/

Retourne la liste des mots extraits de CHAINE en utilisant les espaces comme délimiteurs de mots. On peut le considérer comme équivalent à :

```
split(' ', q/CHAINE/);
```

avec comme différence que la liste est générée lors de la compilation, et que, dans un contexte scalaire, c'est le dernier élément de la liste qui est retourné. Donc <l'expression :>

```
qw(foo bar baz)
```

est sémantiquement équivalente à la liste :

```
'foo', 'bar', 'baz'
```

Quelques exemples fréquemment rencontrés :

```
use POSIX qw(setlocale localeconv)
@EXPORT = qw(foo bar baz);
```

Une erreur assez commune consiste à séparer les mots par des virgules ou à mettre des commentaires dans une qw-chaîne multi-lignes. C'est la raison pour laquelle use warnings ou l'option -w (c'est à dire la variable \$^W) produit un message d'avertissement lorsque CHAINE contient le caractère "," ou le caractère "#".

### s/MOTIF/REPLACEMENT/egimosx

Recherche le motif dans une chaîne puis, s'il est trouvé, le substitue par le texte de REPLACEMENT et retourne finalement le nombre de substitutions effectuées. Sinon, renvoie faux (en l'espèce, la chaîne vide).

Si aucune chaîne n'est spécifiée via =~ ou !~, la recherche et la substitution s'appliquent à la variable \$\_. (La chaîne spécifiée par =~ doit être une variable scalaire, un élément d'un tableau ou d'une table de hachage ou une affectation de l'un ou de l'autre... en un mot, une lvalue.)

Si le délimiteur choisi est l'apostrophe, aucune interpolation n'est effectuée ni sur MOTIF ni sur REPLACEMENT. Sinon, si MOTIF contient un \$ qui ressemble plus à une variable qu'à un test de fin de chaîne, la variable sera interpolée dans le motif lors de l'exécution. Si vous voulez que le motif ne soit compilé qu'une seule fois la première fois que la variable est interpolée, utilisez l'option /o. Si l'évaluation du motif est la chaîne nulle, la dernière expression rationnelle reconnue est utilisée à la place. Voir *perlre* pour de plus amples informations à ce sujet. Voir *perllocale* pour des informations supplémentaires qui s'appliquent lors de l'usage de use locale.

Les options sont :

- e Évaluez la partie droite comme une expression.
- g Substitution globale, c.-à-d. toutes les occurrences.
- i Motif indépendant de la casse (majuscules/minuscules).
- m Traitement de chaîne comme étant multi-lignes.
- o Compilation du motif uniquement la première fois.
- s Traitement de la chaîne comme étant une seule ligne.
- x Utilisation des expressions rationnelles étendues.

N'importe quel délimiteur (ni blanc ni alphanumérique) peut remplacer les barres obliques (slash). Si l'apostrophe est utilisée, aucune interpolation n'est effectuée sur la chaîne de remplacement (par contre, le modificateur /e passe outre). Au contraire de Perl 4, Perl 5 considère l'accent grave (backtick) comme un délimiteur normal ; le texte de

remplacement n'est pas évalué comme une commande. Si le MOTIF est délimité par des caractères fonctionnant en paire (comme les parenthèses), le texte de REMPLACEMENT a sa propre paire de délimiteurs qui peuvent être ou non des caractères fonctionnant par paire, par ex. `s(foo)(bar)` ou `s<foo>/bar/`. L'option `/e` implique que la partie remplacement sera interprétée comme une expression Perl à part entière et donc évaluée comme telle. Par contre, sa validité syntaxique est évaluée lors de la compilation. Une seconde option `e` provoquera l'évaluation de la partie REMPLACEMENT avant son exécution en tant qu'expression Perl.

Exemples:

```
s/\bgreen\b/mauve/g; # ne modifie pas wintergreen

$path =~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # motif dynamique

($foo = $bar) =~ s/this/that/; # recopie puis modification

$count = ($paragraph =~ s/Mister\b/Mr./g); # calcul du nombre de substitution

$_ = 'abc123xyz';
s/\d+/$&*2/e; # produit 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e; # produit 'abc 246xyz'
s/\w/$& x 2/eg; # produit 'aabbcc 224466xxyyzz'

s/%(.)/$percent{$1}/g; # pas de /e
s/%(.)/$percent{$1} || $&/ge; # une expression donc /e
s/^(\\w+)/&pod($1)/ge; # appel de fonction

expansion des variables dans $_, mais dynamique uniquement
en utilisant le déréférencement symbolique
s/^(\\w+)/${$1}/g;

Ajoute un à chaque nombre présent dans la chaîne
s/(\\d+)/1 + $1/eg;

Ceci développe toutes les variables scalaires incluses
(même les lexicales) dans $_ : $1 est tout d'abord interpolé
puis évalué
s/(\\$\\w+)/$1/eeg;

Suppression des commentaires C (presque tous).
$program =~ s {
 /* # Reconnais le début du commentaire
 .*? # Reconnais un minimum de caractères
 */ # Reconnais la fin de commentaire
} []gsx;

s/^\s*(.*?)\s*$/$1/; # suppression des espaces aux extrémités de $_ (couteux)

for ($variable) { # suppression des espaces aux extrémités de $variable (efficace)
 s/^\s+//;
 s/\s+$//;
}

s/([^]*) *([^]*)/$2 $1/; # inverse les deux premiers champs
```

Remarquez l'usage de `$` à la place de `\` dans le dernier exemple. À l'inverse de `sed`, nous utilisons la forme `<digit>` uniquement dans la partie gauche. Partout ailleurs, c'est `$<digit>`.

Dans certains cas, il ne suffit pas de mettre `/g` pour modifier toutes les occurrences. Voici quelques cas communs :

```
placer des virgules correctement dans un entier
(NdT: en français, on mettrait des espaces)
1 while s/(.*\d)(\d\d\d)/$1,$2/g; # perl4
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g; # perl5

remplacement des tabulations par 8 espaces
1 while s/\t+/' ' x (length($&)*8 - length('$')%8)/e;
```

`tr/LISTERECHEE/LISTEREMPLACEMENT/cds`

**y/LISTERECHERCHEE/LISTEREMPLACEMENT/cds**

Translittère chacune des occurrences des caractères de la liste recherchée par le caractère correspondant de la liste de remplacement. Cette opérateur retourne le nombre de caractères remplacés ou supprimés. Si aucune chaîne n'est spécifié via `~` ou `!`, la translittération s'applique à `$_`. (La chaîne spécifiée par `~` doit être une variable scalaire, un élément d'un tableau ou d'un table de hachage ou une affectation de l'un ou de l'autre... en un mot, une lvalue.)

Un intervalle de caractères peut-être spécifié grâce à un tiret. Donc `tr/A-J/0-9/` effectue les mêmes translittérations que `tr/ACEGIBDFHJ/0246813579/`. Pour les adeptes de `sed`, `y` est fourni comme un synonyme de `tr`. Si la liste recherchée est délimitée par des caractères fonctionnant par paire (comme les parenthèses) alors la liste de remplacement a ses propres délimiteurs, par ex. `tr[A-Z][a-z]` ou `tr(+\-*//)ABCD/`.

Notez que `tr` ne reconnaît pas les classes de caractères des expressions rationnelles telles que `\d` ou `[:lower:]`. L'opérateur `tr` n'est donc pas équivalent à l'utilitaire `tr(1)`. Si vous devez effectuer des translittérations majuscules/minuscules, voyez du côté de `lc` in *perlfunc* et `uc` in *perlfunc*, et, en général, du côté de l'opérateur `s` si vous avez besoin des expressions rationnelles.

Remarquez aussi que le concept d'intervalle de caractères n'est pas vraiment portable entre différents codages – et même dans un même codage, cela peut produire un résultat que vous n'attendez pas. Un bon principe de base est de n'utiliser que des intervalles qui commencent et se terminent dans le même alphabet ([a-e], [A-E]) ou dans les chiffres ([0-9]). Tout le reste n'est pas sûr. Dans le doute, énumérez l'ensemble des caractères explicitement.

Les options (ou modificateurs) :

- `c` Complémente la SEARCHLIST.
- `d` Efface les caractères trouvés mais non remplacés.
- `s` Agrège les caractères de remplacement dupliqués.

Si le modificateur `/c` est utilisé, c'est le complément de la liste recherchée qui est utilisé. Si le modificateur `/d` est spécifié, tout caractère spécifié dans `LISTERECHERCHEE` et sans équivalent dans `LISTEREMPLACEMENT` est effacé. (Ceci est tout de même plus flexible que le comportement de certains programme `tr` qui efface tout ce qui est dans `LISTERECHERCHEE`, point!) Si le modificateur `/s` est spécifié, les suites de caractères qui sont translittérés par le même caractère sont agrégées en un seul caractère.

Si le modificateur `/d` est utilisé, `LISTEREMPLACEMENT` est toujours interprété exactement comme spécifié. Sinon, si `LISTEREMPLACEMENT` est plus court que `LISTERECHERCHEE`, le dernier caractère est répété autant de fois que nécessaire pour obtenir la même longueur. Si `LISTEREMPLACEMENT` est vide, `LISTERECHERCHEE` est utilisé à la place. Ce dernier point est très pratique pour comptabiliser les occurrences d'une classe de caractères ou pour agréger les suites de caractères d'une classe.

Exemples :

```
$ARGV[1] =~ tr/A-Z/a-z/; # tout en minuscule

$cnt = tr/*/*/; # compte les étoiles dans $_

$cnt = $sky =~ tr/*/*/; # compte les étoiles dans $sky

$cnt = tr/0-9//; # compte les chiffres dans $_

tr/a-zA-Z//s; # bookkeeper -> bokeper

($HOST = $host) =~ tr/a-zA-Z/;

tr/a-zA-Z/ /cs; # remplace tous les non alphanumériques
 # par un seul espace

tr [\200-\377]
 [\000-\177]; # efface le 8ème bit.
```

Si plusieurs caractères de translittération sont donnés pour un caractère, seul le premier est utilisé :

```
tr/AAA/XYZ/
```

translittérera tous les A en X.

Remarquez que puisque la table de translittération est construite lors de la compilation, ni `LISTERECHERCHEE` ni `LISTEREMPLACEMENT` ne sont sujettes à l'interpolation de chaîne entre guillemets. Cela signifie que si vous voulez utiliser des variables, vous devez utiliser `eval()` :

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

**<<EOF**

Il existe une syntaxe de saisie de chaînes de caractères sur plusieurs lignes basée sur la syntaxe du "here-document" du shell. À la suite d'un <<, vous placez une chaîne de terminaison du texte et toutes les lignes à partir de celle-ci et jusqu'à cette chaîne de terminaison constitueront votre texte. La chaîne de terminaison peut-être un simple identificateur (un mot) ou un texte entre guillemets ou entre apostrophes. Le choix entre les guillemets ou les apostrophes déterminera si l'interpolation sera active ou non, comme pour les chaînes de caractères classiques. Un identificateur aura le même comportement que si il était entre guillemets (et donc l'interpolation sera active). Il ne doit pas y avoir d'espace entre le << et l'identificateur si il n'est pas entre guillemets ou entre apostrophes. (Si il y a un espace, il sera considéré comme l'identificateur vide, qui est valide, et sera reconnu dès la première ligne vide.) Le texte de terminaison doit apparaître tel quel (sans guillemets ni apostrophes ni aucun espace autour) en tant que ligne finale.

```

 print <<EOF;
The price is $Price.
EOF

 print << "EOF"; # Pareil que ci-dessus
The price is $Price.
EOF

 print << 'EOC'; # exécute les commandes
echo hi there
echo lo there
EOC

 print <<"foo", <<"bar"; # vous pouvez les empiler
I said foo.
foo
I said bar.
bar

 myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT

```

N'oubliez pas de mettre un point-virgule à la fin de votre instruction sinon Perl pourrait croire que vous essayez de faire ce qui suit :

```

 print <<ABC
179231
ABC
+ 20;

```

Si vous souhaitez que vos here-docs soient indentés comme le reste de votre code, vous devrez supprimer vous-même les espaces supplémentaires en début de ligne :

```

($quote = <<'FINIS') =~ s/^\s+//gm;
 The Road goes ever on and on,
 down from the door where it began.
FINIS

```

Si vous utilisez un here-doc à l'intérieur d'une construction utilisant des délimiteurs, comme `s///eg`, votre texte doit commencer après le délimiteur final. Donc, n'écrivez pas :

```

s/this/<<E . 'that'
the other
E
. 'more '/eg;

```

mais écrivez :

```

s/this/<<E . 'that'
. 'more '/eg;
the other
E

```

Si votre chaîne de terminaison est la toute dernière ligne de votre programme, soyez sûr qu'elle se termine par un saut de ligne sinon Perl produira l'avertissement suivant : **Can't find string terminator "END" anywhere before EOF...**

De plus, les règles d'interpolation classiques des chaînes de caractères (comme `q{}`, `qq{}` et autres) ne sont pas applicables à la chaîne de terminaison elle-même sauf celle permettant d'insérer le délimiteur lui-même.

```
print << "abc\`def";
testing...
abc`def
```

Pour finir, la chaîne de terminaison ne peut pas être sur plusieurs lignes. En d'autres termes, ça ne peut être qu'une chaîne littérale. Respectez cela et tout ira bien.

### 29.1.30 Les détails sordides de l'interprétation des chaînes

Face à quelque chose qui pourrait avoir différentes interprétations, Perl utilise le principe du **FCQJP** (qui signifie Fait Ce Que Je Pense) pour choisir l'interprétation la plus probable. Cette stratégie fonctionne tellement bien que les utilisateurs de Perl soupçonnent rarement l'ambiguïté de ce qu'ils écrivent. Par contre, de temps à autre, l'idée que se fait Perl diffère de ce que l'auteur du programme pensait.

L'objet de cette partie est de clarifier la manière dont Perl interprète les chaînes. La raison la plus fréquente pour laquelle quelqu'un a besoin de connaître tous ces détails est l'utilisation d'une expression rationnelle "velue". Par contre, les premiers pas de l'interprétation d'une chaîne sont les mêmes pour toutes les constructions de chaînes. Ils sont donc expliqués ensemble.

L'étape la plus importante de l'interprétation des chaînes dans Perl est la première exposée ci-dessous : lors d'une interprétation de chaînes, Perl cherche d'abord la fin de la construction puis il interprète le contenu de cette construction. Si vous comprenez cette règle, vous pouvez sauter les autres étapes en première lecture. Contrairement à cette première étape, les autres étapes contredisent beaucoup moins souvent les attentes de l'utilisateur.

Quelques-unes des passes exposées plus loin sont effectuées simultanément. Mais comme le résultat est le même, nous les considérerons successivement une par une. Selon la construction, Perl effectue ou non certaines passes (de une à cinq) mais elles sont toujours appliquées dans le même ordre.

#### Trouver la fin

La première passe consiste à trouver la fin de la construction qui peut être la suite de caractères `"\nEOF\n"` d'une construction `<<EOF`, le `/` qui termine une construction `qq/`, le `]` qui termine une construction `qq[` ou le `>` qui termine un fileglob commencé par `<`.

Lors de la recherche d'un caractère délimiteur non appairé comme `/`, les combinaisons comme `\\` et `\/` sont sautées. Lors de la recherche d'un délimiteur appairé comme `]`, les combinaisons `\\`, `\]` et `\[` sont sautées ainsi que les constructions imbriquées `[ ]`. Lors de la recherche d'un délimiteur constitué d'une suite de caractères, rien n'est omis.

Pour les constructions en trois parties (`s///`, `y///` et `tr///`), la recherche est répétée une fois supplémentaire.

Lors de cette recherche, aucune attention n'est accordée à la sémantique de la construction, donc :

```
"$hash{"$foo/$bar"}"
```

ou :

```
m/
bar # Ce n'est PAS un commentaire, ce slash / termine m// !
/x
```

ne constituent donc pas des constructions légales. L'expression se termine au premier `"` ou `/` et le reste apparaît comme une erreur de syntaxe. Remarquez que puisque le slash qui termine `m//` est suivi pas un ESPACE, ce n'est pas une constructions `m//x` mais bien un constructions `m//` sans le modificateur `/x`. Donc `#` est interprété comme un `#` littéral.

Notez aussi que rien n'est fait pour détecter une construction du type `\c\` durant cette recherche. Donc le second `\` dans `qq/\c\` sera vu comme le début de `\/` et le `/` qui suit ne sera pas reconnu comme délimiteur. En remplacement, vous pouvez utiliser `\034` ou `\x1c` à la fin des constructions délimitées.

#### Suppression des barres obliques inverses (backslash) précédents les délimiteurs

Lors de la seconde passe, le texte entre le délimiteur de départ et le délimiteur de fin est recopié à un endroit sûr et le `\` des combinaisons constituées par un `\` suivi du délimiteur (ou des délimiteurs si celui de départ diffère de celui de fin) est supprimé. Cette suppression n'a pas lieu pour des délimiteurs multi-caractères.

Remarquez que la combinaison `\\` est laissée telle quelle.

À partir de ce moment plus aucune information concernant le(s) délimiteur(s) n'est utilisée.

## Interpolation

L'étape suivante est l'interpolation appliquée au texte isolé de ses délimiteurs. Il y a quatre cas différents.

```
<<'EOF',m",s"',tr///,y///
```

Aucune interpolation n'est appliquée.

```
",q//
```

La seule interpolation est la suppression du \ des paires \\.

```
""",",qq//,qx//,<file*glob>
```

Les \Q, \U, \u, \L, \l (éventuellement associé avec \E) sont transformés en leur construction Perl correspondante et donc "\$foo\Qbaz\$bar" est transformé en \$foo . (quotemeta("baz" . \$bar)) en interne. Les autres combinaisons constituées d'un \ suivi d'un ou plusieurs caractères sont remplacées par le ou les caractères appropriés.

Soyez conscient que *tout ce qui est entre \Q et \E* est interpolé de manière classique. Donc "\Q\E" ne contient pas de \E : il contient \Q, \ et E donc le résultat est le même que "\\\E". Plus généralement, la présence de backslash entre \Q et \E aboutit à des résultats contre-intuitifs. Donc "\Q\t\E" est converti en quotemeta("\t") qui est la même chose que "\\\t" (puisque TAB n'est pas alphanumérique). Remarquez aussi que :

```
$str = '\t';
return "\Q$str";
```

est peut-être plus proche de *l'intention* de celui qui écrit "\Q\t\E".

Les scalaires et tableaux interpolés sont convertis en une série d'opérations de concaténation join et .. Donc "\$foo XXX '@arr'" devient :

```
$foo . " XXX '" . (join "$", @arr) . "'";
```

Toutes les étapes précédentes sont effectuées simultanément de la gauche vers la droite.

Puisque le résultat de "\Q STRING \E" a tous ses méta-caractères quotés, il n'y a aucun moyen d'insérer littéralement un \$ ou un @ dans une paire \Q\E : si il est protégé par \, un \$ deviendra "\\\\$" et sinon il sera interprété comme le début d'un scalaire à interpolé.

Remarquez aussi que l'interpolation de code doit décider où se termine un scalaire interpolé. Par exemple "a \$b -> {c}" peut signifier :

```
"a " . $b . " -> {c}";
```

ou :

```
"a " . $b -> {c};
```

Dans la plupart des cas, le choix est de considérer le texte le plus long possible n'incluant pas d'espaces entre ses composants et contenant des crochets ou accolades bien équilibrés. Puisque le résultat peut-être celui d'un vote entre plusieurs estimateurs heuristiques, le résultat n'est pas strictement prévisible. Heureusement, il est habituellement correct pour les cas ambigus.

**?RE?, /RE/, m/RE/, s/RE/fo/,**

Le traitement des \Q, \U, \u, \L, \l et des interpolations est effectué quasiment de la même manière qu'avec les constructions qq// sauf que la substitution d'un \ suivi d'un ou plusieurs caractères spéciaux pour les expressions rationnelles (incluant \) n'a pas lieu. En outre, dans les constructions (?{BLOC}), (?# comment), et #-comment présentes dans les expressions rationnelles //x, aucun traitement n'est effectué. C'est le premier cas ou la présence de l'option //x est significative.

L'interpolation a quelques bizarreries : \$, \$( et \$) ne sont pas interpolés et les constructions comme \$var[QQCH] peuvent être vues (selon le vote de plusieurs estimateurs différents) soit comme un élément d'un tableau soit comme \$var suivi par une alternative RE. C'est là où la notation \${arr[\$bar]} prend tout son intérêt : /\${arr[0-9]}/ est interprété comme l'élément -9 du tableau et pas comme l'expression rationnelle contenu dans \$arr suivie d'un chiffre (qui est l'interprétation de /\$arr[0-9]/). Puisqu'un vote entre différents estimateurs peut avoir lieu, le résultat n'est pas prévisible.

C'est à ce moment que la construction \1 est convertie en \$1 dans le texte de remplacement de s/// pour corriger les incorrigibles adeptes de sed qui n'ont pas encore compris l'idiome. Un avertissement est émis si le pragma use warnings ou l'option -w (c'est à dire la variable \$^W) est actif.

Remarquez que l'absence de traitement de \. crée des restrictions spécifiques sur le texte après traitement : si le délimiteur est /, on ne peut pas obtenir \/ comme résultat de cette étape (/ finirait l'expression rationnelle, \/ serait transformé en / par l'étape précédente et \/ serait laissé tel quel). Puisque / est équivalent à \/ dans une expression rationnelle, ceci ne pose problème que lorsque le délimiteur est un caractère spécial pour le moteur RE comme dans s\*foo\*bar\*, m[foo], ou ?foo? ou si le délimiteur est un caractère alphanumérique comme dans :

```
m m ^ a \s* b mx;
```

Dans l'expression rationnelle ci-dessus qui est volontairement obscure pour l'exemple, le délimiteur est `m`, le modificateur est `mx` et après la suppression des backslash, l'expression est la même que `m/ ^ a \s* b /mx`. Il y a de nombreuses raisons de vous encourager à ne choisir que des caractères non alphanumériques et non blancs comme séparateur.

Cette étape est la dernière pour toutes les constructions sauf pour les expressions rationnelles qui sont traitées comme décrit ci-après.

### Interpolation des expressions rationnelles

Toutes les étapes précédentes sont effectuées lors de la compilation du code Perl alors que celle que nous décrivons ici l'est a priori lors de l'exécution (bien qu'elle soit parfois effectuée lors de la compilation pour des raisons d'optimisation). Après tous les pré-traitements précédents (et évaluation des éventuels concaténations, jointures, changements de casse et applications de `quotemeta()` déduits), la chaîne résultante est transmise au moteur RE pour compilation.

Ce qui se passe à l'intérieur du moteur RE est bien mieux expliqué dans *perlre* mais dans un souci de continuité, nous l'exposons un peu ici.

Voici donc une autre étape où la présence du modificateur `//x` est prise en compte. Le moteur RE explore la chaîne de gauche à droite et la convertit en un automate à états finis.

Les caractères "backslashés" sont alors remplacés par la chaîne correspondante (comme pour `\{`) ou génèrent des noeuds spéciaux dans l'automate à états finis (comme pour `\b`). Les caractères ayant un sens spécial pour le moteur RE (comme `|`) génèrent les noeuds ou les groupes de noeuds correspondants. Les commentaires (`?#...`) sont ignorés. Tout le reste est converti en chaîne littérale à reconnaître ou est ignoré (par exemple les espaces et les commentaires `#` si le modificateur `//x` est présent).

Remarquez que le traitement de la construction `[...]` est effectué en utilisant des règles complètement différentes du reste de l'expression rationnelle. Le terminateur de cette construction est trouvé en utilisant les mêmes règles que celles utilisées pour trouver le terminateur d'une construction délimitée (comme `{}`). La seule exception est le `]` qui suit immédiatement le `[` et qui est considéré comme précédé d'un backslash. De manière similaire, la construction `{?{...}}` n'est explorée que pour vérifier que les parenthèses, crochets et autres accolades sont bien équilibrés.

Il est possible d'inspecter la chaîne envoyée au moteur RE ainsi que l'automate à états finis résultant. Voir les arguments `debug/debugcolor` de la directive `use re` et/ou l'option Perl **-Dr** dans Options de Ligne de Commande in *perlrun*.

### Optimisation des expressions rationnelles

Cette étape n'est citée que dans un souci de complétude. Puisque elle ne change en rien la sémantique, les détails de cette étape ne sont pas documentés et sont susceptibles d'évoluer. Cette étape est appliquée à l'automate à états finis généré lors des étapes précédentes.

C'est lors de cette étape que `split()` optimise silencieusement `/^/` en `/^/m`.

#### 29.1.31 Opérateurs d'E/S

Il y a plusieurs opérateurs d'E/S (Entrée/Sortie) que vous devez connaître.

Une chaîne entourée d'apostrophes inversées (accents graves) subit tout d'abord une substitution des variables exactement comme une chaîne entre guillemets. Elle est ensuite interprétée comme une commande et la sortie de cette commande est la valeur du pseudo-littéral comme avec un shell. Dans un contexte scalaire, la valeur retournée est une seule chaîne constituée de toutes les lignes de la sortie. Dans un contexte de liste, une liste de valeurs est retournée, chacune des ces valeurs contenant une ligne de la sortie. (Vous pouvez utiliser `$/` pour utiliser un terminateur de ligne différent.) La commande est exécutée à chaque fois que le pseudo-littéral est évalué. La valeur du statut de la commande est retournée dans `$?` (voir *perlvar* pour l'interprétation de `$?`). À l'inverse de **csh**, aucun traitement n'est appliqué aux valeurs retournées – les passages à la ligne restent des passages à la ligne. À l'inverse de la plupart des shells, les apostrophes inversées n'empêchent pas l'interprétation des noms de variables dans la commande. Pour passer littéralement un `$` au shell, vous devez le protéger en le préfixant par un backslash (barre oblique inversée). La forme générale des apostrophes inversées est `qx//`. (Puisque les apostrophes inversées impliquent toujours un passage par l'interprétation du shell, voir *perlsec* pour tout ce qui concerne la sécurité.)

Dans un contexte scalaire, évaluer un filehandle entre supérieur/inférieur produit le ligne suivante de ce fichier (saut à la ligne inclus si il y a lieu) ou `undef` à la fin du fichier. Lorsque `$/` a pour valeur `undef` (c.-à-d. le mode *file slurp*) et que le fichier est vide, `"` est retourné lors de la première lecture puis ensuite `undef`.

Normalement vous devez affecter cette valeur à une variable mais il y a un cas où une affectation automagique a lieu. Si et seulement si cette opérateur d'entrée est la seule chose présente dans la condition d'une boucle `while` ou `for(;;)` alors



la valeur est automatiquement affectée à la variable `$_`. (Cela peut vous sembler bizarre, mais vous utiliserez de telles constructions dans la plupart des scripts Perl que vous écrirez.) La variable `$_` n'est pas implicitement localisée. Vous devrez donc le faire explicitement en mettant `local $_;`

Les lignes suivantes sont toutes équivalentes :

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

et celle-ci a un comportement similaire mais sans utiliser `$_` :

```
while (my $line = <STDIN>) { print $line }
```

Dans ces constructions, la valeur affectée (que ce soit automatiquement ou explicitement) est ensuite testée pour savoir si elle est définie. Ce test de définition évite les problèmes avec des lignes qui ont une valeur qui pourrait être interprétée comme fausse par perl comme par exemple "" ou "0" sans passage à la ligne derrière. Si vous voulez réellement tester la valeur de la ligne pour terminer votre boucle, vous devrez la tester explicitement :

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

Dans tous les autres contextes booléens, `<filehandle>` sans un test explicite de définition (par `defined`) déclenchera un message d'avertissement si le pragma `use warnings` ou l'option `-w` (c'est à dire la variable `$^W`) est actif.

Les filehandles `STDIN`, `STDOUT`, et `STDERR` sont prédéfinis. (Les filehandles `stdin`, `stdout`, et `stderr` fonctionnent aussi exceptés dans les packages où ils sont interprétés comme des identifiants locaux.) Des filehandles supplémentaires peuvent être créés par la fonction `open()`. Voir *perlopentut* et *open* in *perlfunc* pour plus de détails.

Si `<FILEHANDLE>` est utilisé dans un contexte de liste, une liste constituée de toutes les lignes est retournée avec une ligne par élément de la liste. Il est facile d'utiliser de grande quantité de mémoire par ce moyen. Donc, à utiliser avec précaution.

`<FILEHANDLE>` peut aussi être écrit `readline(*FILEHANDLE)`. Voir *readline* in *perlfunc*.

Le filehandle vide `<>` est spécial et peut être utilisé pour émuler le comportement de `sed` et de `awk`. L'entrée de `<>` provient soit de l'entrée standard soit de tous les fichiers listés sur la ligne de commande. Voici comment ça marche : la première fois que `<>` est évalué, on teste le tableau `@ARGV` et s'il est vide alors `$ARGV[0]` est positionné à "-" qui lorsqu'il sera ouvert lira l'entrée standard. Puis le tableau `@ARGV` est traité comme une liste de nom de fichiers. La boucle :

```
while (<>) {
 ... # code pour chaque ligne
}
```

est équivalent au pseudo-code Perl suivant :

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
 open(ARGV, $ARGV);
 while (<ARGV>) {
 ... # code pour chaque ligne
 }
}
```

sauf qu'il est moins volumineux et qu'il marche réellement. Il décale vraiment le tableau `@ARGV` et stocke le nom du fichier courant dans la variable `$ARGV`. Il utilise aussi en interne le filehandle `ARGV` - `<>` est simplement un synonyme de `<ARGV>` qui est magique. (Le pseudo-code précédent ne fonctionne pas car il tient pas compte de l'aspect magique de `<ARGV>`.)

Vous pouvez modifier `@ARGV` avant la premier `<>` tant que vous y laissez la liste des noms de fichiers que vous voulez. Le numéro de ligne (`$.`) augmente exactement comme si vous aviez un seul gros fichier. Regardez l'exemple de `eof` pour savoir comment le réinitialiser à chaque fichier.

Si vous voulez affecter à `@ARGV` votre propre liste de fichiers, procédez de la manière suivante. La ligne suivante positionne `@ARGV` à tous les fichiers de type texte si aucun `@ARGV` n'est fourni :

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

Vous pouvez même les positionner avec des commandes pipe (tube). Par exemple, la ligne suivante applique automatiquement **gzip** aux arguments compressés :

```
@ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

Si vous voulez passer des options à votre script, vous pouvez utiliser l'un des modules `Getopts` ou placer au début de votre script une boucle du type :

```
while ($_ = $ARGV[0], /^-/) {
 shift;
 last if /^--$/;
 if (/^-D(.*)/) { $debug = $1 }
 if (/^-v/) { $verbose++ }
 # ... # autres options
}

while (<>) {
 # ... # code pour chaque ligne
}
```

Le symbole `<>` retournera `undef` à la fin des fichiers une seule fois. Si vous l'appellez encore une fois après, il supposera que vous voulez traiter une nouvelle liste `@ARGV` et, si vous n'avez pas rempli `@ARGV`, il traitera l'entrée `STDIN`.

Si la chaîne entre supérieur/inférieur est une simple variable scalaire (par ex. `<$foo>`) alors cette variable doit contenir le nom du filehandle à utiliser comme entrée ou son `typeglob` ou une référence vers un `typeglob`. Par exemple :

```
$fh = *STDIN;
$line = <$fh>;
```

Si ce qui est entre supérieur/inférieur n'est ni un filehandle, ni une simple variable scalaire contenant un nom de filehandle, un `typeglob` ou une référence à un `typeglob`, il est alors interprété comme un motif de nom de fichier à appliquer et ce qui est retourné est soit la liste de tous les noms de fichiers ou juste le nom suivant dans la liste selon le contexte. La distinction n'est faite que par la syntaxe. Cela signifie que `<$x>` est toujours la lecture d'une ligne à partir d'un handle indirect mais que `<$hash{key}>` est toujours un motif de nom de fichiers. Tout cela parce que `$x` est une simple variable scalaire alors que `$hash{key}` ne l'est pas – c'est un élément d'une table de hachage. Même `<$x >` (notez l'espace supplémentaire) sera traité comme `glob("$x ")` et non comme `readline($x)`.

Comme pour les chaînes entre guillemets, un niveau d'interprétation est appliqué au préalable mais vous ne pouvez pas dire `<$foo>` parce que c'est toujours interprété comme un filehandle indirect (explications du paragraphe précédent). (Dans des versions antérieures de Perl, les programmeurs inséraient des accolades pour forcer l'interprétation comme un motif de nom de fichier: `<${foo}>`. De nos jours, il est considéré plus propre d'appeler la fonction interne explicitement par `glob($foo)` qui est probablement la meilleure façon de faire dans le premier cas.) Exemple :

```
while (<*.c>) {
 chmod 0644, $_;
}
```

est équivalent à :

```
open(F00, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<F00>) {
 chomp;
 chmod 0644, $_;
}
```

sauf que la recherche des fichiers est faite en interne par l'extension standard `File::Glob`. Bien sûr, le moyen le plus court d'obtenir le résultat précédent est :

```
chmod 0644, <*.c>;
```

Un motif de noms de fichier évalue ses arguments uniquement lorsqu'il débute une nouvelle liste. Toutes les valeurs doivent être lues avant de recommencer. Dans un contexte de liste, cela n'a aucune importance puisque vous récupérez toutes les valeurs quoi qu'il arrive. Dans un contexte scalaire, par contre, l'opérateur retourne la valeur suivante à chaque fois qu'il est appelé ou la valeur `undef` à la fin. Comme pour les filehandles un `defined` automatique est généré lorsque l'opérateur est utilisé comme test d'un `while` ou d'un `for` - car sinon certains noms de fichier légaux (par ex. un fichier nommé `0`) risquent de terminer la boucle. Encore une fois, `undef` n'est retourné qu'une seule fois. Donc si vous n'attendez qu'une seule valeur, il vaut mieux écrire :

```
($file) = <blurch*>;
```

que :

```
$file = <blurch*>;
```

car dans le dernier cas vous aurez soit un nom de fichier soit faux.

Si vous avez besoin de l'interpolation de variables, il est définitivement meilleur d'utiliser la fonction `glob()` parce que l'ancienne notation peut être confondu avec la notation des filehandles indirects.

```
@files = glob("$dir/*. [ch]");
@files = glob($files[$i]);
```

### 29.1.32 Traitement des constantes

Comme en C, Perl évalue un certain nombre d'expressions lors de la compilation lorsqu'il peut déterminer que tous les arguments d'un opérateur sont statiques et n'ont aucun effet de bord. En particulier, la concaténation de chaînes a lieu lors de la compilation entre deux littéraux qui ne sont pas soumis à l'interpolation de variables. L'interprétation du backslash (barre oblique inversée) a lieu elle aussi lors de la compilation. Vous pouvez dire :

```
'Now is the time for all' . "\n" .
'good men to come to.'
```

qui sera réduit à une seule chaîne de manière interne. Vous pouvez aussi dire :

```
foreach $file (@filenames) {
 if (-s $file > 5 + 100 * 2**16) { }
}
```

le compilateur pré-calculera la valeur représentée par l'expression et l'interpréteur n'aura plus à le faire.

### 29.1.33 No-ops (opération vide)

Perl n'a pas officiellement d'opérateur no-op (une opération vide) mais les constantes `0` et `1` sont des cas spéciaux évitant de produire un avertissement si elles sont utilisées dans un contexte vide si bien que vous pouvez écrire :

```
1 while foo();
```

### 29.1.34 Opérateurs bit à bit sur les chaînes

Les chaînes de bits de longueur arbitraire peuvent être manipulées par les opérateurs bit à bit (`~` | `&` `^`).

Si les opérandes d'un opérateur bit à bit sont des chaînes de longueurs différentes, les opérateurs `|` et `^` agiront comme si l'opérande le plus court était complété par des bits à zéro à droite alors que l'opérateur `&` agira comme si l'opérande le plus long était tronqué à la longueur du plus court. Notez que la granularité pour de telles extensions ou troncatures est d'un ou plusieurs octets.

```
Exemples ASCII
print "j p \n" ^ " a h"; # affiche "JAPH\n"
print "JA" | " ph\n"; # affiche "japh\n"
print "japh\nJunk" & '_____'; # affiche "JAPH\n";
print 'p N$' ^ " E<H\n"; # affiche "Perl\n";
```

Si vous avez l'intention de manipuler des chaînes de bits, vous devez être certain de fournir des chaînes de bits : si un opérande est un nombre cela implique une opération bit à bit **numérique**. Vous pouvez explicitement préciser le type d'opération que vous attendez en utilisant "" ou 0+ comme dans les exemples ci-dessous :

```
$foo = 150 | 105; # produit 255 (0x96 | 0x69 vaut 0xFF)
$foo = '150' | 105; # produit 255
$foo = 150 | '105'; # produit 255
$foo = '150' | '105'; # produit la chaîne '155' (si en ASCII)

$baz = 0+$foo & 0+$bar; # les deux opérandes explicitement numériques
$biz = "$foo" ^ "$bar"; # les deux opérandes explicitement chaînes
```

Voir `vec` in *perlfunc* pour savoir comment manipuler des bits individuellement dans un vecteur de bits.

### 29.1.35 Arithmétique entière

Par défaut Perl suppose qu'il doit effectuer la plupart des calculs arithmétiques en virgule flottante. Mais en disant :

```
use integer;
```

vous dites au compilateur qu'il peut utiliser des opérations entières (si il le veut) à partir de ce point et jusqu'à la fin du bloc englobant. Un BLOC interne peut contredire cette commande en disant :

```
no integer;
```

qui durera jusqu'à la fin de ce BLOC. Notez que cela ne signifie pas que tout et n'importe quoi devient entier mais seulement que Perl peut utiliser des opérations entières si il le veut. Par exemple, même avec `use integer. sqrt(2)` donnera toujours quelque chose comme 1.4142135623731.

Utilisés sur des nombres, les opérations bit à bit ("&", "|", "^", "~", "<<", et ">>") produisent toujours des résultats entiers. (Mais voir aussi Opérateurs bit à bit sur les chaînes.) Par contre, `use integer` a encore une influence sur eux. Par défaut, leurs résultats sont interprétés comme des entiers non-signés. Par contre, si `use integer` est actif, leurs résultats sont interprétés comme des entiers signés. Par exemple, `~0` est habituellement évalué comme un grande valeur entière. Par contre, `use integer; ~0` donne -1 sur des machines à complément à deux.

### 29.1.36 Arithmétique en virgule flottante

Bien que `use integer` propose une arithmétique uniquement entière, il n'y a aucun moyen similaire d'imposer des arrondis ou des troncatures à un certain nombre de décimales. Pour arrondir à un nombre de décimales précis, la méthode la plus simple est d'utiliser `sprintf()` ou `printf()`.

Les nombres en virgule flottante ne sont qu'une approximation de ce que les mathématiciens appellent les nombres réels. Il y a infiniment plus de réels que de flottants donc il faut arrondir quelques angles. Par exemple :

```
printf "%.20g\n", 123456789123456789;
produit 123456789123456784
```

Tester l'égalité ou l'inégalité exacte de nombres flottants n'est pas une bonne idée. Voici un moyen (relativement coûteux) pour tester l'égalité de deux nombres flottants sur un nombre particulier de décimales. Voir le volume II de Knuth pour un traitement plus robuste de ce sujet.

```
sub fp_equal {
 my ($X, $Y, $POINTS) = @_ ;
 my ($tX, $tY);
 $tX = sprintf("%.${POINTS}g", $X);
 $tY = sprintf("%.${POINTS}g", $Y);
 return $tX eq $tY;
}
```

Le module POSIX (qui fait partie de la distribution standard de perl) implémente `ceil()`, `floor()` et un certain nombre d'autres fonctions mathématiques et trigonométriques. Le module `Math::Complex` (qui fait partie de la distribution standard de perl) définit de nombreuses fonctions mathématiques qui fonctionnent aussi bien sur des nombres réels que sur des nombres complexes. `Math::Complex` n'est pas aussi performant que POSIX mais POSIX ne peut pas travailler avec des nombres complexes.

Arrondir dans une application financière peut avoir de sérieuses implications et la méthode d'arrondi utilisée doit être spécifiée précisément. Dans ce cas, il peut être plus sûr de ne pas faire confiance aux différents systèmes d'arrondi proposés par Perl mais plutôt d'implémenter vous-mêmes la fonction d'arrondi dont vous avez besoin.

### 29.1.37 Les grands nombres

Les modules standard `Math::BigInt` et `Math::BigFloat` fournissent des variables arithmétiques en précision infinie et redéfinissent les opérateurs correspondants. Au prix d'un peu d'espace et de beaucoup de temps, ils permettent d'éviter les embûches classiques associées aux représentations en précision limitée.

```
use Math::BigInt;
$x = Math::BigInt->new('123456789123456789');
print $x * $x;

affiche +15241578780673678515622620750190521
```

Il existe plusieurs modules vous permettant d'effectuer vos calculs avec une précision arbitraire ou illimitée (limitée uniquement par le temps de calcul et la mémoire disponible). Il existe aussi des modules non standard qui fournissent des implémentations plus rapides passant par des bibliothèques C externes.

En voici une courte liste non-exhaustive :

<code>Math::Fraction</code>	grand rationnels exacts comme 9973 / 12967
<code>Math::String</code>	traite les chaînes comme des nombres
<code>Math::FixedPrecision</code>	calcul avec précision arbitraire
<code>Math::Currency</code>	pour les calculs financiers
<code>Bit::Vector</code>	manipulation rapide de vecteurs de bits (en C)
<code>Math::BigIntFast</code>	Extension de <code>Bit::Vector</code> pour les grands nombres
<code>Math::Pari</code>	donne accès à la bibliothèque C Pari
<code>Math::BigInteger</code>	utilise une bibliothèque C externe
<code>Math::Cephes</code>	utilise la bibliothèque C Cephes (pas de grands nombres)
<code>Math::Cephes::Fraction</code>	rationnels via la bibliothèque Cephes
<code>Math::GMP</code>	encore une autre bibliothèque C externe

Il ne vous reste plus qu'à choisir.

## 29.2 TRADUCTION

### 29.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 29.2.2 Traducteur

Traduction initiale et mise à jour vers 5.8.8 : Paul Gaborit <Paul DOT Gaborit AT enstimac DOT fr>.

### 29.2.3 Relecture

Personne pour l'instant.

# Chapitre 30

## perlsub

Les sous-programmes (ou sous-routines) de Perl

### 30.1 SYNOPSIS

Pour déclarer des sous-programmes :

```
sub NAME; # Une déclaration "en avant".
sub NAME(PROTO); # idem, mais avec des prototypes
sub NAME : ATTRS; # avec attributs
sub NAME(PROTO) : ATTRS; # avec attributs et prototypes

sub NAME BLOCK # Une déclaration et une définition.
sub NAME(PROTO) BLOCK # idem, mais avec des prototypes
sub NAME : ATTRS BLOCK # avec attributs
sub NAME(PROTO) : ATTRS BLOCK # avec attributs et prototypes
```

Pour définir un sous-programme anonyme lors de l'exécution :

```
$subref = sub BLOCK; # pas de prototype
$subref = sub (PROTO) BLOCK; # avec prototype
$subref = sub : ATTRS BLOCK; # avec attributs
$subref = sub (PROTO) : ATTRS BLOCK; # avec proto et attributs
```

Pour importer des sous-programmes :

```
use MODULE qw(NAME1 NAME2 NAME3);
```

Pour appeler des sous-programmes :

```
NAME(LIST); # & est optionnel avec les parenthèses.
NAME LIST; # Les parenthèses sont optionnelles si le
 # sous-programme est prédéclaré ou importé.
&NAME(LIST); # Court-circuite les prototypes.
&NAME; # Rend la @_ courante visible par le
 # sous-programme appelé.
```

## 30.2 DESCRIPTION

Comme beaucoup de langages, Perl fournit des sous-programmes définis par l'utilisateur. Ils peuvent se trouver n'importe où dans le programme principal. Ils peuvent être chargés depuis d'autres fichiers via les mots-clés `do`, `require` ou `use`, ou être générés à la volée en utilisant `eval` ou des sous-programmes anonymes. Vous pouvez même appeler une fonction indirectement en utilisant une variable contenant son nom ou une référence à du CODE.

Le modèle de Perl pour l'appel de fonction et le renvoi de valeurs est simple : tous les paramètres sont passés aux fonctions comme une simple liste de scalaires, et toutes les fonctions renvoient de la même manière à leur appelant une simple liste de scalaires. Tout tableau ou hachage dans ces listes d'appel et de retour s'effondreront, perdant leur identité – mais vous pouvez toujours utiliser le passage par référence à la place pour éviter cela. Les listes d'appel ou de retour peuvent contenir autant d'éléments scalaires que vous le désirez (une fonction sans instruction de retour explicite est souvent appelée un sous-programme, mais il n'y a vraiment pas de différence du point de vue de Perl).

Tous les arguments passés à la routine se retrouvent dans le tableau `@_`. Ainsi, si vous appelez une fonction avec deux arguments, ceux-ci seront stockés dans `$_[0]` et `$_[1]`. Le tableau `@_` est un tableau local, mais ses éléments sont des alias pour les véritables paramètres scalaires. En particulier, si un élément `$_[0]` est mis à jour, l'argument correspondant est mis à jour (ou bien une erreur se produit s'il n'est pas modifiable). Si un argument est un élément de tableau ou de hachage qui n'existait pas lors de l'appel de la fonction, cet élément est créé seulement lorsque (et si) il est modifié ou si on y fait référence (certaines versions précédentes de Perl créaient l'élément qu'il soit ou non affecté). L'affectation à la totalité du tableau `@_` retire les alias, et ne met à jour aucun argument.

Une instruction `return` peut être utilisée pour sortir d'un sous-programme, en spécifiant éventuellement une valeur de retour, qui sera évaluée dans le contexte approprié (liste, scalaire ou vide) selon le contexte d'appel du sous-programme. Si vous ne spécifiez aucune valeur de retour, le sous-programme retourne une liste vide dans un contexte de liste, la valeur indéfinie dans un contexte scalaire et rien du tout dans un contexte vide. Si vous retournez un ou plusieurs agrégats (tableau ou table de hachage), ils seront tous aplatis en une seule grande liste.

Si aucun `return` n'est trouvé et si la dernière instruction est une expression, sa valeur est retournée. Si la dernière instruction est une structure de contrôle de boucle comme `foreach` ou `while`, la valeur retournée est non spécifiée. Un sous-programme vide retourne une liste vide.

Perl n'a pas de paramètres formels nommés. En pratique tout ce que vous avez à faire est d'affecter à une liste `my()` les contenant. Les variables que vous utilisez dans la fonction et qui ne sont pas déclarées comme privées sont des variables globales. Pour des détails sanglants sur la création de variables privées, voir Variables privées via `my()` (§30.2.1) et Valeurs temporaires via `local()` (§30.2.3). Pour créer des environnements protégés pour un ensemble de fonctions dans un paquetage séparé (et probablement un fichier séparé), voir Paquetages in *perlmod*.

Exemple :

```
sub max {
 my $max = shift(@_);
 foreach $foo (@_) {
 $max = $foo if $max < $foo;
 }
 return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Exemple :

```
obtient une ligne, en y combinant les lignes la continuant qui
débutent par un blanc

sub get_line {
 $thisline = $lookahead; # Variables globales !!
 LINE: while (defined($lookahead = <STDIN>)) {
 if ($lookahead =~ /^[\t]/) {
 $thisline .= $lookahead;
 }
 else {
 last LINE;
 }
 }
 return $thisline;
}
```

```
$lookahead = <STDIN>; # obtient la première ligne
while (defined($line = get_line()) {
 ...
}
```

Affectez à une liste de variables privées pour nommer vos arguments :

```
sub maybeaset {
 my($key, $value) = @_;
 $Foo{$key} = $value unless $Foo{$key};
}
```

Puisque l'affectation copie les valeurs, ceci a aussi pour effet de transformer l'appel par référence en appel par valeur. Autrement, une fonction est libre de faire des modifications de @\_ sur place et de changer les valeurs de son appelant.

```
upcase_in($v1, $v2); # ceci change $v1 et $v2
sub upcase_in {
 for (@_) { tr/a-z/A-Z/ }
}
```

Vous n'avez pas le droit de modifier les constantes de cette façon, bien sûr. Si un argument était en vérité un littéral et que vous essayiez de le changer, vous lèveriez une exception (probablement fatale). Par exemple, ceci ne fonctionnera pas :

```
upcase_in("frederick");
```

Ce serait bien plus sûr si la fonction upcase\_in() était écrite de façon à renvoyer une copie de ses paramètres au lieu de les changer sur place :

```
($v3, $v4) = upcase($v1, $v2); # cela ne change pas $v1 et $v2
sub upcase {
 return unless defined wantarray; # contexte vide, ne fait rien
 my @parms = @_;
 for (@parms) { tr/a-z/A-Z/ }
 return wantarray ? @parms : $parms[0];
}
```

Notez comment cette fonction (non prototypée) ne se soucie pas qu'on lui ait passé de vrais scalaires ou des tableaux. Perl voit tous les arguments comme une grosse et longue liste plate de paramètres dans @\_. C'est un domaine dans lequel brille le style simple de passage d'arguments de Perl. La fonction upcase() fonctionnerait parfaitement bien sans avoir à changer la définition de upcase() même si nous lui donnions des choses telles que ceci :

```
@newlist = upcase(@list1, @list2);
@newlist = upcase(split /:/, $var);
```

Ne vous laissez toutefois pas tenter de faire :

```
(@a, @b) = upcase(@list1, @list2);
```

Tout comme la liste plate de paramètres entrante, la liste de retour est aussi aplatie. Donc tout ce que vous êtes parvenu à faire ici est de tout stocker dans @a et de vider @b. Voir [Passage par référence](#) pour savoir comment faire autrement.

Un sous-programme peut être appelé en utilisant un préfixe & explicite. Le & est optionnel en Perl moderne, tout comme le sont les parenthèses si le sous-programme a été prédéclaré. Le & n'est *pas* optionnel lorsque l'on nomme juste le sous-programme, comme lorsqu'il est utilisé en tant qu'argument de defined() ou de undef(). Il n'est pas non plus optionnel lorsque vous voulez faire un appel indirect de sous-programme avec le nom d'un sous-programme ou une référence utilisant les constructions &\$subref() ou &{\$subref}(), bien que la notation \$subref->() résoud ce problème. Voir [perlref](#) pour plus de détails à ce sujet.

Les sous-programmes peuvent être appelés de façon récursive. Si un sous-programme est appelé en utilisant la forme &, la liste d'arguments est optionnelle, et si elle est omise, aucun tableau @\_ n'est mis en place pour le sous-programme : le tableau @\_ au moment de l'appel est visible à la place dans le sous-programme. C'est un mécanisme d'efficacité que les nouveaux utilisateurs pourraient préférer éviter.



```

&foo(1,2,3); # passe trois arguments
foo(1,2,3); # idem

foo(); # passe une liste nulle
&foo(); # idem

&foo; # foo() obtient les arguments courants, comme foo(@_) !!
foo; # comme foo() SI le sous-programme foo est
 # prédéclaré, sinon "foo"

```

Non seulement la forme & rend la liste d'arguments optionnelle, mais elle désactive aussi toute vérification de prototype sur les arguments que vous fournissez. C'est en partie pour des raisons historiques, et en partie pour avoir une façon pratique de tricher si vous savez ce que vous êtes en train de faire. Voir *Prototypes* ci-dessous.

Les sous-programmes dont les noms sont entièrement en majuscules sont réservées pour le noyau de Perl, tout comme les modules dont les noms sont entièrement en minuscules. Une fonction entièrement en majuscules est une convention informelle signifiant qu'elle sera appelée indirectement par le système d'exécution lui-même, habituellement en réponse à un événement. Les sous-programmes qui font des choses spéciales prédéfinies sont entre autres AUTOLOAD, CLONE et DESTROY – plus toutes les fonctions mentionnées dans *perlite* et *PerlIO::via*.

Les sous-programmes BEGIN, CHECK, INIT et END sont plutôt des blocs spéciaux, de code, nommés, qui peuvent apparaître plusieurs fois dans un paquetage et qu'il vous est **impossible** d'appeler explicitement. Voir BEGIN, CHECK, INIT et END in *perlmod*.

### 30.2.1 Variables privées via my()

Synopsis:

```

my $foo; # déclare $foo locale lexicalement
my (@wid, %get); # déclare une liste de variables locale
my $foo = "flurp"; # déclare $foo lexical, et l'initialise
my @oof = @bar; # déclare @oof lexical, et l'initialise
my $x : Foo = $y; # similaire, avec application d'un attribut

```

**AVERTISSEMENT** : l'usage de listes d'attributs sur les déclarations my est expérimental. La sémantique précise et l'interface associée sont sujettes à changement. Voir *attributes* et *Attribute::Handlers*.

L'opérateur my déclare que les variables listées doivent être confinées lexicalement au bloc où elles se trouvent, dans la conditionnelle (if/unless/elsif/else), la boucle (for/foreach/while/until/continue), le sous-programme, l'eval, ou le fichier exécuté, requis ou utilisé via do/require/use. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses. Tous les éléments listés doivent être des lvalues légaux. Seuls les identifiants alphanumériques peuvent avoir une portée lexicale – pour l'instant!, les identifiants magiques prédéfinis comme \$/ peuvent, à la place, être localisés par local.

Contrairement aux variables dynamiques créées par l'opérateur local, les variables lexicales déclarées par my sont totalement cachées du monde extérieur, y compris de tout sous-programme appelé. Cela est vrai s'il s'agit du même sous-programme appelé depuis lui-même ou depuis un autre endroit – chaque appel obtient sa propre copie.

Ceci ne veut pas dire qu'une variable my() déclarée dans une portée lexicale englobante statiquement serait invisible. Seules les portées dynamiques sont rétrécies. Par exemple, la fonction bumpx() ci-dessous a accès à la variable lexicale \$x car le my et le sub se sont produits dans la même portée, probablement la portée du fichier.

```

my $x = 10;
sub bumpx { $x++ }

```

Un eval(), toutefois, peut voir les variables lexicales de la portée dans laquelle il est évalué, tant que les noms ne sont pas cachés par des déclarations à l'intérieur de l'eval() lui-même. Voir *perlref*.

La liste de paramètres de my() peut être affectée si on le désire, ce qui vous permet d'initialiser vos variables (si aucune valeur initiale n'est donnée à une variable particulière, celle-ci est créée avec la valeur indéfinie). Ceci est utilisé couramment pour nommer les paramètres d'entrée d'un sous-programme. Exemples :

```

$args = "fred"; # variable "globale"
$n = cube_root(27);
print "$arg thinks the root is $n\n";
fred thinks the root is 3

```

```
sub cube_root {
 my $arg = shift; # le nom n'importe pas
 $arg **= 1/3;
 return $arg;
}
```

Le `my` est simplement un modificateur sur quelque chose que vous pourriez affecter. Donc lorsque vous affectez les variable dans sa liste d'arguments, le `my` ne change pas le fait que ces variables soient vues comme un scalaire ou un tableau. Donc

```
my ($foo) = <STDIN>; # FAUX ?
my @FOO = <STDIN>;
```

les deux fournissent un contexte de liste à la partie droite, tandis que

```
my $foo = <STDIN>;
```

fournit un contexte scalaire. Mais ce qui suit ne déclare qu'une seule variable :

```
my $foo, $bar = 1; # FAUX
```

Cela a le même effet que

```
my $foo;
$bar = 1;
```

La variable déclarée n'est pas introduite (n'est pas visible) avant la fin de l'instruction courante. Ainsi,

```
my $x = $x;
```

peut être utilisé pour initialiser une nouvelle variable `$x` avec la valeur de l'ancienne `$x`, et l'expression

```
my $x = 123 and $x == 123
```

est fausse à moins que l'ancienne variable `$x` ait la valeur 123.

Les portées lexicales de structures de contrôle ne sont pas précisément liées aux accolades qui délimitent le bloc qu'elles contrôlent ; les expressions de contrôle font aussi partie de cette portée. Ainsi, dans la boucle

```
while (my $line = <>) {
 $line = lc $line;
} continue {
 print $line;
}
```

la portée de `$line` s'étend à partir de sa déclaration et sur tout le reste de la boucle (y compris la clause `continue`), mais pas au-delà. De façon similaire, dans la conditionnelle

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
 user_agrees();
} elsif ($answer =~ /^no$/i) {
 user_disagrees();
} else {
 chomp $answer;
 die "'$answer' is neither 'yes' nor 'no'";
}
```

la portée de `$answer` s'étend à partir de sa déclaration et sur tout le reste de la conditionnelle, y compris les clauses `elsif` et `else`, s'il y en a, mais pas au-delà. Voir Instructions simples in *perlsyn* pour plus d'informations sur la portée des variables dans des instructions avec modificateurs.

La boucle `foreach` a pour défaut de donner une portée dynamiquement à sa variable d'index à la manière de `local`. Toutefois, si la variable d'index est préfixée par le mot-clé `my`, ou s'il existe déjà un lexical ayant ce nom dans la portée, alors un nouveau lexical est créé à la place. Ainsi dans la boucle

```
for my $i (1, 2, 3) {
 some_function();
}
```

la portée de `$i` s'étend jusqu'à la fin de la boucle, rendant la valeur de `$i` inaccessible dans `some_function()`.

Certains utilisateurs pourraient désirer encourager l'usage de variables à portée lexicale. Comme aide pour intercepter les utilisations implicites de variables de paquetage, qui sont toujours globales, si vous dites

```
use strict 'vars';
```

alors toute variable mentionnée à partir de là jusqu'à la fin du bloc doit soit se référer à une variable lexicale, soit être prédéclarée via `our` ou `use vars`, soit encore être totalement qualifiée avec le nom du paquetage. Autrement, une erreur de compilation se produit. Un bloc interne pourrait contrer ceci par `no strict 'vars'`.

Un `my` a un effet à la fois à la compilation et lors de l'exécution. Lors de la compilation, le compilateur le remarque. La principale utilité de ceci est de faire taire `use strict 'vars'`, mais c'est aussi essentiel pour la génération de fermetures telle que détaillée dans *perlref*. L'initialisation effective est toutefois retardée jusqu'à l'exécution, de façon à ce qu'elle soit exécutée au moment approprié, par exemple à chaque fois qu'une boucle traversée.

Les variables déclarées avec `my` ne font pas partie d'un paquetage et ne sont par conséquent jamais totalement qualifiée avec le nom du paquetage. En particulier, vous n'avez pas le droit d'essayer de rendre lexicale une variable de paquetage (ou toute autre variable globale) :

```
my $pack::var; # ERREUR ! Syntaxe Illégale
my $_; # aussi illégal (pour le moment)
```

En fait, une variable dynamique (aussi connue sous le nom de variable de paquetage ou variable globale) est toujours accessible en utilisant la notation totalement qualifiée `::` : même lorsqu'un lexical de même nom est lui aussi visible :

```
package main;
local $x = 10;
my $x = 20;
print "$x and $::x\n";
```

Ceci affichera `20` et `10`.

Vous pouvez déclarer les variables `my` dans la portée extrême d'un fichier pour cacher tous ces identifiants du monde extérieur à ce fichier. Cela est similaire en esprit aux variables statiques de C quand elles sont utilisées au niveau du fichier. Faire cela avec un sous-programme requiert l'usage d'une fermeture (une fonction anonyme qui accède aux lexicaux qui l'entourent). Si vous voulez créer un sous-programme privé qui ne peut pas être appelé en-dehors de ce bloc, il peut déclarer une variable lexicale contenant une référence anonyme de sous-programme :

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

Tant que la référence n'est jamais renvoyée par aucune fonction du module, aucun module extérieur ne peut voir le sous-programme, car son nom n'est dans la table de symboles d'aucun paquetage. Souvenez-vous qu'il n'est jamais *VRAIMENT* appelé `$some_pack::secret_version` ou quoi que ce soit ; c'est juste `$secret_version`, non qualifié et non qualifiable.

Ceci ne fonctionne pas avec les méthodes d'objets, toutefois ; toutes les méthodes d'objets doivent se trouver dans la table de symboles d'un paquetage quelconque pour être trouvées. Voir *Modèles de Fonctions* in *perlref* pour une méthode permettant de contourner ceci.

### 30.2.2 Variables privées persistantes

Le fait qu'une variable lexicale ait une portée lexicale (aussi appelée statique) égale au bloc qui la contient, à l'éval, ou au FICHER `do`, ne veut pas dire que cela fonctionne à l'intérieur d'une fonction comme une variable statique en C. Cela marche plutôt comme une variable `auto` de C, mais avec un nettoyage de la mémoire (un ramasse-miettes) implicite.

Contrairement aux variables locales en C ou en C++, les variables lexicales de Perl ne sont pas nécessairement recyclées juste parce qu'on est sorti de leur portée. Si quelque chose de plus permanent est toujours conscient du lexical, il restera. Tant que quelque chose d'autre fait référence au lexical, ce lexical ne sera pas libéré – ce qui est comme il se doit. Vous

ne voudriez pas que la mémoire soit libérée tant que vous n'avez pas fini de l'utiliser, ou gardée lorsque vous en avez terminé. Le ramasse-miette automatique s'occupe de cela pour vous.

Ceci veut dire que vous pouvez passer en retour ou sauver des références à des variables lexicales, tandis que retourner un pointeur sur un auto en C est une grave erreur. Cela nous donne aussi un moyen de simuler les variables statiques de fonctions C. Voici un mécanisme donnant à une fonction des variables privées de portée lexicale ayant une durée de vie statique. Si vous désirez créer quelque chose ressemblant aux variables statiques de C, entourez juste toute la fonction d'un niveau de bloc supplémentaire, et mettez la variable statique hors de la fonction mais dans le bloc.

```
{
 my $secret_val = 0;
 sub gimme_another {
 return ++$secret_val;
 }
}
$secret_val devient désormais impossible à atteindre pour le
monde extérieur, mais garde sa valeur entre deux appels à
gimme_another
```

Si cette fonction est chargée par `require` ou `use` depuis un fichier séparé, alors cela marchera probablement très bien. Si tout se trouve dans le programme principal, vous devrez vous arranger pour que le `my` soit exécuté tôt, soit en mettant tout le bloc avant votre programme principal, ou, de préférence, en plaçant simplement un `BEGIN` devant lui pour vous assurer qu'il soit exécuté avant que votre programme ne démarre :

```
BEGIN {
 my $secret_val = 0;
 sub gimme_another {
 return ++$secret_val;
 }
}
```

Voir `BEGIN`, `CHECK`, `INIT` et `END` in *perlmod* pour en savoir plus sur la mise en oeuvre des blocks spéciaux `BEGIN`, `CHECK`, `INIT` et `END`.

S'ils sont déclarés au niveau de la plus grande portée (celle du fichier), alors les lexicaux fonctionnent un peu comme les variables statiques de fichier en C. Ils sont disponibles pour toutes les fonctions déclarées en dessous d'eux dans le même fichier, mais sont inaccessibles hors du fichier. Cette stratégie est parfois utilisée dans les modules pour créer des variables privées que la totalité du module peut voir.

### 30.2.3 Valeurs temporaires via `local()`

**AVERTISSEMENT** : en général, vous devriez utiliser `my` au lieu de `local`, parce que c'est plus rapide et plus sûr. Les exceptions à cela incluent les variables globales dont le nom est un caractère de ponctuation, les handles de fichier globaux et les formats globaux et la manipulation directe de la table de symboles de Perl. On utilise `local` lorsqu'on souhaite que la valeur courante d'une variable soit visible par les sous-programmes appelés.

Synopsis :

```
local-isation de valeurs

local $foo; # rend $foo dynamiquement locale
local (@wid, %get); # rend locales toutes les variables d'une liste
local $foo = "flurp"; # rend $foo dynamique, et l'initialise
local @oof = @bar; # rend @oof dynamique, et l'initialise

local $hash{key} = "val"; # rend local la valeur lié à cette clé
local ($cond ? $v1 : $v2); # la plupart des types de lvalues
 # acceptent la local-isation

local-isation de symboles
```

```

local *FH; # rend locales $FH, @FH, %FH, &FH...
local *merlyn = *randal; # maintenant $merlyn est vraiment $randal,
 # @merlyn est vraiment @randal, etc.
local *merlyn = 'randal'; # IDEM : 'randal' est promu *randal
local *merlyn = \$randal; # alias juste $merlyn, pas @merlyn etc

```

Un `local()` modifie ses variables listées pour qu'elle soient "locales" au bloc courant, à l'éval, ou au FICHER `do` – et à *tout sous-programme appelé depuis l'intérieur de ce bloc*. Un `local()` donne juste des valeurs temporaires aux variables globales (au paquetage). Il ne crée *pas* une variable locale. Ceci est connu sous le nom de portée dynamique. La portée lexicale est créé par `my`, qui fonctionne plus comme les déclarations `auto` de C.

Différents types de lvalues peuvent être localisés : les éléments ou les tranches de tableaux ou de tables de hachage, éventuellement de manière conditionnelle (dans la mesure où le résultat est bien localisable), et les références symboliques. Comme pour de simples variables, cela crée de nouvelles valeurs avec une portée dynamique.

Si plus d'une variable est fournie à `local`, elles doivent être placées entre parenthèses. Cet opérateur fonctionne en sauvegardant les valeurs courantes de ces variables dans sa liste d'arguments sur une pile cachée et les restaure à la sortie du bloc, du sous-programme ou de l'éval. Ceci veut dire que les sous-programmes appelés peuvent aussi référencer les variables locales, mais pas les globales. La liste d'arguments peut être affectée si l'on veut, ce qui vous permet d'initialiser vos variables locales (si aucun initialiseur n'est donné pour une variable locale particulière, elle est créée avec la valeur indéfinie).

Puisque `local` est une commande réalisée lors de la phase d'exécution, elle est donc exécutée chaque fois que l'on traverse une boucle. Par conséquent il est toujours plus efficace de localiser vos variables en dehors de la boucle.

### Note grammatical sur `local()`

Un `local` est simplement un modificateur d'une expression donnant une lvalue. Lorsque vous affectez une variable localisée, le `local` ne change pas selon que sa liste est vue comme un scalaire ou un tableau. Donc

```

local($foo) = <STDIN>;
local @FOO = <STDIN>;

```

fournissent tous les deux un contexte de liste à la partie droite, tandis que

```

local $foo = <STDIN>;

```

fournit un contexte scalaire.

### Localisation des variables spéciales

Si vous localisez une variable spéciale, vous lui donnez une nouvelle valeur mais sans lui faire perdre ses fonctionnalités magiques. Cela signifie que tous les effets secondaires de cette magie ont lieu avec cette nouvelle valeur.

Cela permet le fonctionnement d'un code tel que celui-ci :

```

Lit tout le contenu de FILE dans $slurp
{ local $/ = undef; $slurp = <FILE>; }

```

Notez, en revanche, que cela empêche la localisation de certaines variables. Par exemple, l'instruction suivante déclenchera, dans perl 5.9.0, un appel à 'die' avec l'erreur *Modification of a read-only value attempted* puisque `$!` est une variable magique en lecture seule :

```

local $! = 2;

```

De manière similaire mais moins évidente, le code suivant déclenchera un 'die' en perl 5.9.0 :

```

sub f { local $_ = "foo"; print }
for ($!) {
 # now $_ is aliased to $!, thus is magic and readonly
 f();
}

```

Voir la section suivante pour savoir comment palier cela.

**ATTENTION** : la localisation de tableaux ou de tables de hachage liés (par `tie()`) ne fonctionne pas comme cela. Cela sera corrigé dans une prochaine version de Perl. En attendant, évitez de localiser des tableaux ou des tables de hachage liés (la localisation de éléments individuels fonctionnent). Voir *Localisation de tableaux et de tables de hachage liés (par `tie()`) in perl58delta* pour plus de détails.

## Localisation de globs

La construction

```
local $name;
```

créé une nouvelle entrée dans la table de symbole associé au glob `name` dans le paquetage courant. Cela signifie que toutes les variables attachées à cette entrée (`$name`, `@name`, `%name`, `&name` et le handle de fichier `name`) sont dynamiquement réinitialisées.

En particulier, si vous voulez une nouvelle valeur pour le scalaire par défaut `$_`, en évitant le problème potentiel évoqué plus haut lorsque `$_` était attaché à une valeur magique. vous devriez utiliser `local *$_` au lieu de `local $_`.

## Localisation des éléments d'un type composé

Une note sur `local()` et les types composés est nécessaire. Quelque chose comme `local(%foo)` fonctionne en plaçant temporairement un hachage tout neuf dans la table des symboles. L'ancien hachage est mis de côté, mais est caché "derrière" le nouveau.

Ceci signifie que l'ancienne variable est complètement invisible via la table des symboles (i.e. l'entrée de hachage dans le `typeglob *foo`) pendant toute la durée de la portée dynamique dans laquelle le `local()` a été rencontré. Cela a pour effet de permettre d'occulter temporairement toute magie sur les types composés. Par exemple, ce qui suit altèrera brièvement un hachage lié à une autre implémentation :

```
tie %ahash, 'APackage';
[...]
{
 local %ahash;
 tie %ahash, 'BPackage';
 [.. le code appelé verra %ahash lié à 'BPackage'..]
 {
 local %ahash;
 [..%ahash est une hachage normal (non lié) ici..]
 }
}
[..%ahash revient à sa nature liée initiale..]
```

Autre exemple, une implémentation personnelle de `%ENV` pourrait avoir l'air de ceci :

```
{
 local %ENV;
 tie %ENV, 'MyOwnEnv';
 [..faites vos propres manipulations fantaisistes de %ENV ici..]
}
[..comportement normal de %ENV ici..]
```

Il vaut aussi la peine de prendre un moment pour expliquer ce qui se passe lorsque vous localisez un membre d'un type composé (i.e. un élément de tableau ou de hachage). Dans ce cas, l'élément est localisé *par son nom*. Cela veut dire que lorsque la portée du `local()` se termine, la valeur sauvee sera restaurée dans l'élément du hachage dont la clé a été nommée dans le `local()`, ou dans l'élément du tableau dont l'index a été nommé dans le `local()`. Si cet élément a été effacé pendant que le `local()` faisait effet (e.g. par un `delete()` dans un hachage ou un `shift()` d'un tableau), il reprendra vie, étendant potentiellement un tableau et remplissant les éléments intermédiaires avec `undef`. Par exemple, si vous dites

```
%hash = ('This' => 'is', 'a' => 'test');
@ary = (0..5);
{
 local($ary[5]) = 6;
 local($hash{'a'}) = 'drill';
 while (my $e = pop(@ary)) {
 print "$e . . .\n";
 }
}
```

```

 last unless $e > 3;
 }
 if (@ary) {
 $hash{'only a'} = 'test';
 delete $hash{'a'};
 }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash), ".\n";
print "The array has ", scalar(@ary), " elements: ",
 join(' ', map { defined $_ ? $_ : 'undef' } @ary), "\n";

```

Perl affichera

```

6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5

```

Le comportement de local() sur des membres inexistants de types composites est sujet à changements à l'avenir.

### 30.2.4 Lvalue et sous-programme

**AVERTISSEMENT** : les sous-programmes en partie gauche (en lvalue) sont encore expérimentaux et leur implémentation est sujette à changements dans les futures versions de Perl.

Il est possible de retourner une valeur modifiable depuis un sous-programme. Pour ce faire, vous devez déclarer que le sous-programme retourne une lvalue.

```

my $val;
sub canmod : lvalue {
 # return $val; cela ne fonction pas, ne dites pas "return"
 $val;
}

sub nomod {
 $val;
}

canmod() = 5; # modifie $val
nomod() = 5; # ERREUR

```

Le contexte scalaire ou de liste pour le sous-programme et la partie droite de l'affectation est déterminé comme si l'appel au sous-programme était remplacé par un scalaire. Par exemple, si l'on considère :

```
data(2,3) = get_data(3,4);
```

Les deux sous-programmes sont appelés ici dans un contexte scalaire, tandis que dans :

```
(data(2,3)) = get_data(3,4);
```

et dans :

```
(data(2),data(3)) = get_data(3,4);
```

tous les sous-programmes sont appelés dans un contexte de liste.

### Les sous-programmes lvalue sont expérimentaux

Cela semble pratique mais il y a plusieurs raisons pour rester circonspect.

Vous ne devez pas utiliser le mot-clé `return` et vous devez passer la valeur de sortie avant qu'elle soit hors de portée (voir le commentaire dans l'exemple plus haut). Ce n'est pas habituellement un problème mais cela empêche tout de même un `return` explicite dans une boucle qui est pourtant parfois bien pratique.

Cela ne respecte pas l'encapsulation. Un mutateur normal vérifie la valeur fournie avant de l'affecter à l'attribut qu'il protège. Un sous-programme lvalue ne permet pas cela. Considérez le code suivant :

```
my $some_array_ref = []; # protégé par un mutateur ??

sub set_arr { # mutateur normal
 my $val = shift;
 die("expected array, you supplied ", ref $val)
 unless ref $val eq 'ARRAY';
 $some_array_ref = $val;
}
sub set_arr_lv : lvalue { # mutateur par lvalue
 $some_array_ref;
}

set_arr_lv ne peut pas empêcher cela !
set_arr_lv() = { a => 1 };
```

### 30.2.5 Passage d'entrées de table de symbole (typeglobs)

**AVERTISSEMENT** : le mécanisme décrit dans cette section était à l'origine la seule façon de simuler un passage par référence dans les anciennes versions de Perl. Même si cela fonctionne toujours très bien dans les versions modernes, le nouveau mécanisme de référencement est généralement plus facile à utiliser. Voir plus bas.

Parfois, vous ne voulez pas passer la valeur d'un tableau à un sous-programme mais plutôt son nom, pour que le sous-programme puisse modifier sa copie globale plutôt que de travailler sur une copie locale. En perl, vous pouvez vous référer à tous les objets d'un nom particulier en préfixant ce nom par une étoile : `*foo`. Cela est souvent connu sous le nom de "typeglob", car l'étoile devant peut être conçue comme un caractère générique correspondant à tous les drôles de caractères préfixant les variables et les sous-programmes et le reste.

Lorsqu'il est évalué, le typeglob produit une valeur scalaire qui représente tous les objets portant ce nom, y compris tous les handles de fichiers, les formats, ou les sous-programmes. Quand on l'affecte, le nom mentionné se réfère alors à la valeur `*` qui lui a été affectée, quelle qu'elle soit. Exemple :

```
sub doubleary {
 local(*someary) = @_;
 foreach $elem (@someary) {
 $elem *= 2;
 }
}
doubleary(*foo);
doubleary(*bar);
```

Les scalaires sont déjà passés par référence, vous pouvez donc modifier les arguments scalaires sans utiliser ce mécanisme en vous référant explicitement à `$_[0]`, etc. Vous pouvez modifier tous les éléments d'un tableau en passant tous les éléments comme des scalaires, mais vous devez utiliser le mécanisme `*` (ou le mécanisme de référencement équivalent) pour effectuer des `push`, des `pop`, ou changer la taille d'un tableau. Il sera certainement plus rapide de passer le typeglob (ou la référence).

Même si vous ne voulez pas modifier un tableau, ce mécanisme est utile pour passer des tableaux multiples dans une seule LIST, car normalement le mécanisme LIST fusionnera toutes les valeurs de tableaux et vous ne pourrez plus en extraire les tableaux individuels. Pour plus de détails sur les typeglobs, voir *Typeglobs et handles de fichiers* in *perldata*.

### 30.2.6 Quand faut-il encore utiliser local()

Malgré l'existence de `my`, il y a encore trois endroits où l'opérateur `local` brille toujours. En fait, en ces trois endroits, vous devez utiliser `local` à la place de `my`.



1. Vous avez besoin de donner une valeur temporaire à une variable globale, en particulier \$\_.

Les variables globales, comme @ARGV ou les variables de ponctuation, doivent être localisées avec local(). Ce bloc lit dans /etc/motd, et le découpe en morceaux, séparés par des lignes de signes égal, qui sont placés dans @Fields.

```
{
 local @ARGV = ("/etc/motd");
 local $/ = undef;
 local $_ = <>;
 @Fields = split /\s*+=+\s*$/;
}
```

Il est important en particulier de localiser \$\_ dans toute routine qui l'affecte. Surveillez les affectations implicites dans les conditionnelles while.

2. Vous avez besoin de créer un handle de fichier ou de répertoire local, ou une fonction locale.

Une fonction ayant besoin de son propre handle de fichier doit utiliser local() sur le typeglob complet. Ceci peut être utilisé pour créer de nouvelles entrées dans la table des symboles :

```
sub ioqueue {
 local (*READER, *WRITER); # pas my !
 pipe (READER, WRITER) or die "pipe: $!";
 return (*READER, *WRITER);
}
($head, $tail) = ioqueue();
```

Voir le module Symbol pour une façon de créer des entrées de table de symboles anonymes.

Puisque l'affectation d'une référence à un typeglob crée un alias, ceci peut être utilisé pour créer ce qui est effectivement une fonction locale, ou au moins un alias local.

```
{
 local *grow = \&shrink; # seulement jusqu'à la fin de
 # l'existence de ce bloc
 grow(); # appelle en fait shrink()
 move(); # si move() grandit, il rétrécit
 # aussi
}
grow(); # récupère le vrai grow()
```

Voir Modèles de fonctions in *perlref* pour plus de détails sur la manipulation des fonctions par leur nom de cette manière.

3. Vous voulez changer temporairement un seul élément d'un tableau ou d'un hachage.

Vous pouvez localiser juste un élément d'un agrégat. Habituellement, cela est fait dynamiquement :

```
{
 local $SIG{INT} = 'IGNORE';
 funct(); # impossible à interrompre
}
la possibilité d'interrompre est restaurée ici
```

Mais cela fonctionne aussi sur les agrégats déclarés lexicalement. Avant la version 5.005, cette opération pouvait parfois mal se conduire.

### 30.2.7 Passage par référence

Si vous voulez passer plus d'un tableau ou d'un hachage à une fonction – ou les renvoyer depuis elle – de façon qu'ils gardent leur intégrité, vous allez devoir alors utiliser un passage par référence explicite. Avant de faire cela, vous avez besoin de comprendre les références telles qu'elles sont détaillées dans *perlref*. Cette section n'aura peut-être pas beaucoup de sens pour vous sans cela.

Voici quelques exemples simples. Tout d'abord, passons plusieurs tableaux à une fonction puis faisons-lui faire des pop sur eux, et retourner une nouvelle liste de tous leurs derniers éléments :

```
@tailings = popmany (\@a, \@b, \@c, \@d);
```

```

sub popmany {
 my $aref;
 my @retlist = ();
 foreach $aref (@_) {
 push @retlist, pop @$aref;
 }
 return @retlist;
}

```

Voici comment vous pourriez écrire une fonction retournant une liste des clés apparaissant dans tous les hachages qu'on lui passe :

```

@common = inter(\%foo, \%bar, \%joe);
sub inter {
 my ($k, $href, %seen); # locals
 foreach $href (@_) {
 while ($k = each %$href) {
 $seen{$k}++;
 }
 }
 return grep { $seen{$_} == @_ } keys %seen;
}

```

Jusque là, nous utilisons juste le mécanisme normal de retour d'une liste. Que se passe-t-il si vous voulez passer ou retourner un hachage ? Eh bien, si vous n'utilisez que l'un d'entre eux, ou si vous ne voyez pas d'inconvénient à ce qu'ils soient concaténés, alors la convention d'appel normale est valable, même si elle coûte un peu cher.

C'est ici que les gens commencent à avoir des problèmes :

```

(@a, @b) = func(@c, @d);
ou
(%a, %b) = func(%c, %d);

```

Cette syntaxe ne fonctionnera tout simplement pas. Elle définit juste @a ou %a et vide @b ou %b. De plus, la fonction n'a pas obtenu deux tableaux ou hachages séparés : elle a eu une longue liste dans @\_, comme toujours.

Si vous pouvez vous arranger pour que tout le monde règle cela par des références, cela fait du code plus propre, même s'il n'est pas très joli à regarder (phrase louche, NDT). Voici une fonction qui prend deux références de tableau comme arguments et renvoie les deux éléments de tableau dans l'ordre du nombre d'éléments qu'ils contiennent :

```

($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
 my ($cref, $dref) = @_;
 if (@$cref > @$dref) {
 return ($cref, $dref);
 } else {
 return ($dref, $cref);
 }
}

```

Il s'avère en fait que vous pouvez aussi faire ceci :

```

(*a, *b) = func(\@c, \@d);
print "@a has more than @b\n";
sub func {
 local (*c, *d) = @_;
 if (@c > @d) {
 return (\@c, \@d);
 } else {
 return (\@d, \@c);
 }
}

```

Ici nous utilisons les typeglobs pour faire un aliasing de la table des symboles. C'est un peu subtile, toutefois, et en plus cela ne fonctionnera pas si vous utilisez des variables `my`, puisque seules les variables globales (et les locales en fait) sont dans la table des symboles.

Si vous passez des handles de fichiers, vous pouvez habituellement juste utiliser le typeglob tout seul, comme `*STDOUT`, mais les références de typeglobs fonctionnent aussi. Par exemple :

```
splutter(*STDOUT);
sub splutter {
 my $fh = shift;
 print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN);
sub get_rec {
 my $fh = shift;
 return scalar <$fh>;
}
```

Si vous comptez générer de nouveau handles de fichiers, vous pourriez faire ceci. Faites attention de renvoyer juste le `*FH` tout seul, et pas sa référence.

```
sub openit {
 my $path = shift;
 local *FH;
 return open (FH, $path) ? *FH : undef;
}
```

### 30.2.8 Prototypes

Perl supporte une sorte de vérification des arguments très limitée lors de la compilation, en utilisant le prototypage de fonction. Si vous déclarez

```
sub mypush (@@)
```

alors `mypush()` prendra ses arguments exactement comme `push()` le fait. La déclaration de la fonction doit être visible au moment de la compilation. Le prototype affecte seulement l'interprétation des appels nouveau style de la fonction, où le nouveau style est défini comme n'utilisant pas le caractère `&`. En d'autres termes, si vous l'appellez comme une fonction intégrée, alors elle se comportera comme une fonction intégrée. Si vous l'appellez comme un sous-programme à l'ancienne mode, alors elle se comportera comme un sous-programme à l'ancienne. La conséquence naturelle de cette règle est que les prototypes n'ont pas d'influence sur les références de sous-programmes comme `\&foo` ou sur les appels de sous-programmes indirects comme `&{$subref}` ou `$subref->()`.

Les appels de méthode ne sont pas non plus influencés par les prototypes, car la fonction qui doit être appelée est indéterminée au moment de la compilation, puisque le code exact appelé dépend de l'héritage.

Puisque l'intention de cette caractéristique est principalement de vous laisser définir des sous-programmes qui fonctionnent comme des commandes intégrées, voici des prototypes pour quelques autres fonctions qui se compilent presque exactement comme les fonctions intégrées correspondantes.

Déclarées en tant que	Appelé comme
<code>sub mylink (\$\$)</code>	<code>mylink \$old, \$new</code>
<code>sub myvec (\$\$\$)</code>	<code>myvec \$var, \$offset, 1</code>
<code>sub myindex (\$\$;\$)</code>	<code>myindex &amp;getstring, "substr"</code>
<code>sub mysyswrite (\$\$\$;\$)</code>	<code>mysyswrite \$buf, 0, length(\$buf) - \$off, \$off</code>
<code>sub myreverse (@)</code>	<code>myreverse \$a, \$b, \$c</code>
<code>sub myjoin (\$@)</code>	<code>myjoin ":", \$a, \$b, \$c</code>
<code>sub mypop (\@)</code>	<code>mypop @array</code>
<code>sub mysplICE (\@\$\$@)</code>	<code>mysplICE @array, @array, 0, @pushme</code>
<code>sub mykeys (\%)</code>	<code>mykeys %{\$hashref}</code>
<code>sub myopen (*;\$)</code>	<code>myopen HANDLE, \$name</code>
<code>sub mypipe (**)</code>	<code>mypipe READHANDLE, WRITEHANDLE</code>
<code>sub mygrep (&amp;@)</code>	<code>mygrep { /foo/ } \$a, \$b, \$c</code>
<code>sub myrand (\$)</code>	<code>myrand 42</code>
<code>sub mytime ()</code>	<code>mytime</code>

Tout caractère de prototype précédé d'une barre oblique inverse représente un véritable argument qui doit absolument commencer par ce caractère. La valeur passée comme partie de @\_ sera une référence au véritable argument passé dans l'appel du sous-programme, obtenue en appliquant \ à cet argument.

Vous pouvez aussi transformer d'un seul coup en référence à plusieurs types d'argument en utilisant la notation \[ ] :

```
sub myref (\[$@%&*])
```

qui autorisera les appels suivant à myref() :

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

et le premier argument de myref() sera une référence à un scalaire, un tableau, une table de hachage, du code ou à un glob.

Les caractères de prototype non précédés d'une barre oblique inverse ont une signification spéciale. Tout @ ou % sans barre oblique inverse mange les arguments restants, et force un contexte de liste. Un argument représenté par \$ force un contexte scalaire. Un & requiert un sous-programme anonyme, qui, s'il est passé comme premier argument, ne requiert pas le mot-clé "sub" ni une virgule après lui. Un \* permet au sous-programme d'accepter un bareword, une constante, une expression scalaire, un typeglob, ou une référence à un typeglob à cet emplacement. La valeur sera disponible pour le sous-programme soit comme un simple scalaire, soit (dans les deux derniers cas) comme une référence au typeglob. Si vous désirez toujours convertir de tels arguments en référence de typeglob, utilisez Symbol::qualify\_to\_ref() ainsi :

```
use Symbol 'qualify_to_ref';

sub foo (*) {
 my $fh = qualify_to_ref(shift, caller);
 ...
}
```

Un point-virgule sépare les arguments obligatoires des arguments optionnels. Il est redondant devant @ ou %, qui englobent tout le reste.

Notez comment les trois derniers exemples dans la table ci-dessus sont traités spécialement par l'analyseur. mygrep() est compilé comme un véritable opérateur de liste, myrand() comme un véritable opérateur unaire avec une précedence unaire identique à celle de rand(), et mytime() est vraiment sans arguments, tout comme time(). C'est-à-dire que si vous dites

```
mytime +2;
```

Vous obtiendrez mytime() + 2, et pas mytime(2), qui est la façon dont cela serait analysé sans prototype.

Ce qui est intéressant avec & est que vous pouvez générer une nouvelle syntaxe grâce à lui, pourvu qu'il soit en position initiale :

```
sub try (&@) {
 my($try,$catch) = @_;
 eval { &$try };
 if ($@) {
 local $_ = $@;
 &$catch;
 }
}
sub catch (&) { $_[0] }

try {
 die "phooey";
} catch {
 /phooey/ and print "unphooey\n";
};
```

Cela affiche "unphooey" (Oui, il y a toujours des problèmes non résolus à propos de la visibilité de @\_. J'ignore cette question pour le moment (Mais notez que si nous donnons une portée lexicale à @\_, ces sous-programmes anonymes peuvent agir comme des fermetures... (Mince, est-ce que ceci sonne un peu lispien ? (Peu importe))).

Et voici une réimplémentation de l'opérateur `grep` de Perl :

```
sub mygrep (&@) {
 my $code = shift;
 my @result;
 foreach $_ (@_) {
 push(@result, $_) if &$code;
 }
 @result;
}
```

Certaines personnes préféreraient des prototypes pleinement alphanumériques. Les caractères alphanumériques ont intentionnellement été laissés hors des prototypes expressément dans le but d'ajouter un jour futur des paramètres formels nommés. Le principal objectif du mécanisme actuel est de laisser les programmeurs de modules fournir de meilleurs diagnostics à leurs utilisateurs. Larry estime que la notation est bien compréhensible pour les programmeurs Perl, et que cela n'interférera pas notablement avec le cœur du module, ni ne le rendra plus difficile à lire. Le bruit sur la ligne est visuellement encapsulé dans une petite pilule facile à avaler.

Si vous essayez d'utiliser des séquences alphanumériques dans un prototype, vous aurez droit à un avertissement optionnel - "Illegal character in prototype...". Malheureusement, les anciennes versions de Perl autorisaient l'utilisation de ce type de prototype tant que le début était un prototype valide. L'avertissement actuel pourrait devenir une erreur fatale dans une future version de Perl lorsque la majorité du code erroné aura été corrigé.

Il est probablement meilleur de prototyper les nouvelles fonctions, et de ne pas prototyper rétrospectivement les anciennes. C'est parce que vous devez être particulièrement précautionneux avec les exigences silencieuses de la différence entre les contextes de liste et les contextes scalaires. Par exemple, si vous décidez qu'une fonction devrait prendre juste un paramètre, comme ceci :

```
sub func ($) {
 my $n = shift;
 print "you gave me $n\n";
}
```

et que quelqu'un l'a appelée avec un tableau ou une expression retournant une liste :

```
func(@foo);
func(split /:/);
```

Alors vous venez juste de fournir un `scalar` automatique devant leurs arguments, ce qui peut être plus que surprenant. L'ancien `@foo` qui avait l'habitude de ne contenir qu'une chose n'entre plus. À la place, `func()` se voit maintenant passer un 1 ; c'est-à-dire le nombre d'éléments dans `@foo`. Et le `split` se voit appelé dans un contexte scalaire et commence à gribouiller dans votre liste de paramètres @\_. Aïe !

Tout ceci est très puissant, bien sûr, et devrait être utilisé uniquement avec modération pour rendre le monde meilleur.

### 30.2.9 Fonctions constantes

Les fonctions ayant un prototype égal à `()` sont des candidates potentielles à l'insertion en ligne (inlining, NDT). Si le résultat après optimisation et repliement des constantes est soit une constante soit un scalaire de portée lexicale n'ayant pas d'autre référence, alors il sera utilisé à la place des appels à la fonction faits sans `&`. Les appels réalisés en utilisant `&` ne sont jamais insérés en ligne (Voir *constant.pm* pour une façon aisée de déclarer la plupart des constantes).

Les fonctions suivantes seraient toutes insérées en ligne :

```
sub pi () { 3.14159 } # Pas exact, mais proche.
sub PI () { 4 * atan2 1, 1 } # Aussi exact qu'il
 # puisse être, et
 # inséré en ligne aussi !
```

```

sub ST_DEV () { 0 }
sub ST_INO () { 1 }

sub FLAG_FOO () { 1 << 8 }
sub FLAG_BAR () { 1 << 9 }
sub FLAG_MASK () { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ () { not (0x1B58 & FLAG_MASK) }

sub N () { int(OPT_BAZ) / 3 }

sub FOO_SET () { 1 if FLAG_MASK & FLAG_FOO }

```

En revanche, celles qui suivent ne pourront être insérées en ligne ; parce qu'elles contiennent des choses non constantes dans leur propre portée.

```

sub foo_set () { if (FLAG_MASK & FLAG_FOO) { 1 } }

sub baz_val () {
 if (OPT_BAZ) {
 return 23;
 }
 else {
 return 42;
 }
}

```

Si vous redéfinissez un sous-programme qui pouvait être inséré en ligne, vous obtiendrez un avertissement formel (vous pouvez utiliser cet avertissement pour déterminer si un sous-programme particulier est considéré comme constant ou pas). L'avertissement est considéré comme suffisamment sévère pour ne pas être optionnel, car les invocations précédemment compilées de la fonction utiliseront toujours l'ancienne valeur de cette fonction. Si vous avez besoin de pouvoir redéfinir le sous-programme, vous devez vous assurer qu'il n'est pas inséré en ligne, soit en abandonnant le prototype `()` (ce qui change la sémantique de l'appel, donc prenez garde), soit en contrecarrant le mécanisme d'insertion d'une autre façon, comme par exemple

```

sub not_inlined () {
 23 if $];
}

```

### 30.2.10 Surcharges des fonctions prédéfinies

Beaucoup de fonctions prédéfinies peuvent être surchargées, même si cela ne devrait être essayé qu'occasionnellement et pour une bonne raison. Typiquement, cela pourrait être réalisé par un paquetage essayant d'émuler une fonctionnalité prédéfinie manquante sur un système non Unix.

La surcharge ne peut être réalisée qu'en important le nom depuis un module lors de la compilation – la prédéclaration ordinaire n'est pas suffisante. Toutefois, le pragma `use subs` vous laisse, dans les faits, prédéclarer les sous-programmes via la syntaxe d'importation, et ces noms peuvent ensuite surcharger ceux qui sont prédéfinis :

```

use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }

```

Pour se référer sans ambiguïté à la forme prédéfinie, on peut faire précéder le nom prédéfini par le qualificateur de paquetage spéciale `CORE::`. Par exemple, le fait de dire `CORE::open()` se réfère toujours à la fonction prédéfinie `open()`, même si le paquetage courant a importé d'ailleurs un quelconque autre sous-programme appelé `&open()`. Même si cela a l'air d'un appel de fonction normal, ce n'en est pas un : vous ne pouvez pas en créer de référence, telle que celles que l'incorrect `\&CORE::open` pourrait en produire.

Les modules de bibliothèque ne devraient en général pas exporter de noms prédéfinis comme `open` ou `chdir` comme partie de leur liste `@EXPORT` par défaut, car ceux-ci pourraient s'infiltrer dans l'espace de noms de quelqu'un d'autre et changer la sémantique de façon inattendue. À la place, si le module ajoute ce nom à `@EXPORT_OK`, alors il est possible pour un utilisateur d'importer le nom explicitement, mais pas implicitement. C'est-à-dire qu'il pourrait dire

```
use Module 'open';
```

et cela importerait la surcharge `open`. Mais s'il disait

```
use Module;
```

il obtiendrait les importations par défaut sans les surcharges.

Le mécanisme précédent de surcharge des éléments prédéfinis est restreint, assez délibérément, au paquetage qui réclame l'importation. Il existe une seconde méthode parfois applicable lorsque vous désirez surcharger partout une fonction prédéfinie, sans tenir compte des frontières entre espaces de noms. Cela s'obtient en important un sous-programme dans l'espace de noms spécial `CORE::GLOBAL::`. Voici un exemple qui remplace assez effrontément l'opérateur `glob` par quelque chose qui comprend les expressions rationnelles.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
 my $pkg = shift;
 return unless @_;
 my $sym = shift;
 my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
 $pkg->export($where, $sym, @_);
}

sub glob {
 my $pat = shift;
 my @got;
 local *D;
 if (opendir D, '.') {
 @got = grep /$pat/, readdir D;
 closedir D;
 }
 return @got;
}
1;
```

Et voici comment on pourrait en (ab)user :

```
#use REGlob 'GLOBAL_glob'; # surcharge glob() dans TOUS les
 # espaces de noms

package Foo;
use REGlob 'glob'; # surcharge glob() dans Foo::
 # seulement

print for <^[a-z_]+\.pm\>; # montre tous les modules pragmatiques
```

Le commentaire initial montre un exemple artificiel, voire même dangereux. En surchargeant `glob` globalement, vous forceriez le nouveau (et subversif) comportement de l'opérateur `glob` pour *tous* les espaces de noms, sans la complète coopération des modules qui possèdent ces espaces de noms, et sans qu'ils en aient même connaissance. Naturellement, ceci doit être fait avec d'extrêmes précautions – si jamais cela doit être fait.

L'exemple `REGlob` ci-dessus n'implémente pas tous le support nécessaires pour surcharger proprement l'opérateur `glob` de perl. La fonction intégrée `glob` a des comportements différents selon qu'elle apparaît dans un contexte scalaire ou un contexte de listes, mais ce n'est pas le cas de notre `REGlob`. En fait, beaucoup de fonctions prédéfinies de perl ont de tels comportements sensibles au contexte, et ceux-ci doivent être supportés de façon adéquate par une surcharge correctement écrite. Pour un exemple pleinement fonctionnel de surcharge de `glob`, étudiez l'implémentation de `File::DosGlob` dans la bibliothèque standard.

Lorsque vous surchargez une fonction prédéfinie, votre surcharge devrait être cohérente (si possible) avec la syntaxe native de l'originale. Vous pouvez le faire en utilisant le prototype approprié. Pour obtenir le prototype d'une fonction prédéfinie

surchargeable, utilisez la fonction prototype avec comme argument "CORE::nom\_de\_la\_fonction" (voir prototype in *perlfunc*).

Remarquez que certaines fonctions prédéfinies utilise une syntaxe non représentable par un prototype (par exemple `system` ou `chomp`). Si vous les surchargez, vous ne pourrez pas simuler leur syntaxe d'origine.

Les "fonctions" prédéfinies `do`, `require` et `glob` peuvent elles-aussi être surchargées, mais comme elles sont magiques, elles conservent leur syntaxe d'origine et vous n'avez pas besoin de définir de prototype pour leurs fonctions de remplacement. (Par contre, vous ne pouvez pas surcharger la syntaxe `do BLOC`).

`require` possède en plus un peu de magie noire ; si vous appelez votre fonction `require` de remplacement par `require Foo::Bar`, vous recevrez en fait l'argument "Foo/Bar.pm" dans `@_`. Voir `require` in *perlfunc*.

Et, en se référant à l'exemple précédent, si vous surchargez `glob`, l'opérateur "glob" <<\*> sera aussi surchargé.

De manière similaire, la surcharge de la fonction `readline` surcharge aussi l'opérateur d'E/S équivalent <FILEHANDLE>.

Et pour finir, certaines fonctions prédéfinies (comme `exists` ou `grep`) ne peuvent pas être surchargées.

### 30.2.11 Autochargement

Si vous appelez un sous-programme qui n'est pas défini, vous obtenez ordinairement une erreur fatale immédiate se plaignant que le sous-programme n'existe pas (De même pour les sous-programmes utilisés comme méthodes, lorsque la méthode n'existe dans aucune classe de base du paquetage de la classe). Toutefois, si un sous-programme `AUTOLOAD` est défini dans le paquetage ou dans les paquetages utilisés pour localiser le sous-programme originel, alors ce sous-programme `AUTOLOAD` est appelé avec les arguments qui auraient été passés au sous-programme originel. Le nom pleinement qualifié du sous-programme originel apparaît magiquement dans la variable globale `$AUTOLOAD` du même paquetage que la routine `AUTOLOAD`. Le nom n'est pas passé comme un argument ordinaire parce que, euh, eh bien, juste parce que, c'est pour ça...

La plupart des routines `AUTOLOAD` chargent une définition du sous-programme requis en utilisant `eval()`, puis l'exécutent en utilisant une forme spéciale de `goto()` qui efface la routine `AUTOLOAD` de la pile sans laisser de traces (Voir le module standard `AutoLoader` pour un exemple). Mais une routine `AUTOLOAD` peut aussi juste émuler la routine et ne jamais la définir. Par exemple, supposons qu'une fonction non encore définie doive juste appeler `system` avec ces arguments. Tout ce que vous feriez est ceci :

```
sub AUTOLOAD {
 my $program = $AUTOLOAD;
 $program =~ s/.*:://;
 system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

En fait, si vous prédéclarez les fonctions que vous désirez appeler de cette façon, vous n'avez même pas besoin des parenthèses :

```
use subs qw(date who ls);
date;
who "am", "i";
ls -l;
```

Un exemple plus complet de ceci est le module `Shell` standard, qui peut traiter des appels indéfinis à des sous-programmes comme des appels de programmes externes.

Des mécanismes sont disponibles pour aider les auteurs de modules à découper les modules en fichiers autochargeables. Voir le module standard `AutoLoader` décrit dans *AutoLoader* et dans *AutoSplit*, les modules standards `SelfLoader` dans *SelfLoader*, et le document expliquant comment ajouter des fonctions C au code Perl dans *perlx*.



### 30.2.12 Attributs de sous-programme

Une déclaration ou une définition de sous-programme peut contenir une liste d'attributs qui lui sont associés. Si une telle liste est présente, elle est découpée aux espaces et aux deux-points et traitée comme si un `use attributes` avait été rencontré. Voir *attributes* pour plus de détails sur les différents attributs actuellement supportés. Contrairement aux limitations de l'`use attrs` obsolète, la syntaxe `sub : ATTRLIST` fonctionne pour associer les attributs à une prédéclaration, et pas seulement dans la définition de sous-programme.

Les attributs doivent être des identifiants simples valides (sans aucune ponctuation à part le caractère "\_"). Ils peuvent être suivis d'une liste de paramètres, qui n'est contrôlée que pour voir si ses parenthèses ('(',')') sont correctement imbriquées.

Exemples de syntaxe valide (même si les attributs sont inconnus) :

```
sub fnord (&%) : switch(10,foo(7,3)) : expensive ;
sub plugh () : Ugly('\(") :Bad ;
sub xyzy : _5x5 { ... }
```

Exemples de syntaxe invalide :

```
sub fnord : switch(10,foo() ; # () non équilibrées
sub snoid : Ugly('() ; # () non équilibrées
sub xyzy : 5x5 ; # "5x5" n'est pas un identifiant valide
sub plugh : Y2::north ; # "Y2::north" n'est pas un identifiant simple
sub snurt : foo + bar ; # "+" n'est pas un deux-points ni une espace
```

La liste d'attributs est passée comme une liste de chaînes constantes au code qui les associe au sous-programme. En particulier, le second exemple de syntaxe valide ci-dessus à cette allure en termes d'analyse syntaxique et d'invocation :

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

Pour plus de détails sur les listes d'attributs et leur manipulation, voir *attributes* en *Attributes::Handlers*.

## 30.3 VOIR AUSSI

Voir Modèles de fonctions in *perlref* pour plus de détails sur les références et les fermetures. Voir *perlx*s pour apprendre à appeler des sous-programmes en C depuis Perl. Voir *perlembed* pour apprendre à appeler des sous-programmes Perl depuis C. Voir *perlmod* pour apprendre à emballer vos fonctions dans des fichiers séparés. Voir *perlmodlib* pour apprendre quels modules de bibliothèques sont fournis en standard sur votre système. Voir *perltoot* pour apprendre à faire des appels à des méthodes objet.

## 30.4 TRADUCTION

### 30.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 30.4.2 Traducteur

Traduction : Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 30.4.3 Relecture

Personne pour l'instant.

# Chapitre 31

## perlfunc

Fonctions Perl prédéfinies

### 31.1 DESCRIPTION

Les fonctions de cette section peuvent être utilisées en tant que termes dans une expression. Elles se séparent en deux catégories principales : les opérateurs de listes et les opérateurs unaires nommés. Ceux-ci diffèrent dans leurs relations de priorité avec la virgule qui les suit. (Cf. la table de priorité dans *perlop*.) Les opérateurs de liste prennent plusieurs arguments alors que les opérateurs unaires n'en prennent jamais plus d'un. Une virgule termine alors l'argument d'un opérateur unaire mais sépare les arguments d'un opérateur de liste. Un opérateur unaire fournit en général un contexte scalaire à son argument, alors qu'un opérateur de liste fournit un contexte, soit scalaire, soit de liste, pour ses arguments. S'il propose les deux, les arguments scalaires seront les premiers et la liste d'arguments suivra. (Notez qu'il ne peut y avoir qu'une seule liste d'arguments.) Par exemple, `splice()` a trois arguments scalaires suivis d'une liste alors que `gethostbyname()` a quatre arguments scalaires.

Dans la description syntaxique qui suit, les opérateurs de liste qui attendent une liste (et fournissent un contexte de liste pour les éléments de cette liste) ont pour argument LISTE. Une telle liste peut être constituée de toute combinaison de valeurs d'arguments scalaires ou de listes ; les valeurs de listes seront incluses dans la liste comme si chaque élément individuel était interpolé à cet emplacement de la liste, formant ainsi la valeur d'une longue liste unidimensionnelle. Les éléments de LISTE doivent être séparés par des virgules.

Toute fonction de la liste ci-dessous peut être utilisée avec ou sans parenthèses autour de ses arguments. (Les descriptions syntaxiques les omettent) Si vous utilisez les parenthèses, la simple (mais parfois surprenante) règle est la suivante : ça *RESSEMBLE* à une fonction, donc c'*EST* une fonction, et la priorité importe peu. Sinon, c'est un opérateur de liste ou un opérateur unaire et la priorité a son importance. Les espaces entre la fonction et les parenthèses ne comptent pas, vous devez donc faire parfois très attention :

```
print 1+2+4; # affiche 7.
print(1+2) + 4; # affiche 3.
print (1+2)+4; # affiche aussi 3 !
print +(1+2)+4; # affiche 7.
print ((1+2)+4); # affiche 7.
```

Si vous exécutez Perl avec l'option `-w`, vous pourrez en être averti. Par exemple, la troisième ligne ci-dessus génère :

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

Quelques rares fonctions ne prennent aucun argument et ne sont donc ni des opérateurs unaires ni des opérateurs de liste. Cela inclut des fonctions telles que `time` et `endpwent`. Par exemple, `time+86_400` signifie toujours `time()` + `86_400`.

Pour les fonctions qui peuvent être utilisées dans un contexte scalaire ou de liste, une erreur non fatale est généralement indiquée dans un contexte scalaire en retournant la valeur indéfinie, et dans un contexte de liste en retournant la liste nulle.

Rappelez-vous de l'importante règle suivante : il n'y a **aucune règle** qui lie le comportement d'une expression dans un contexte de liste à son comportement dans un contexte scalaire, et réciproquement. Cela peut générer deux résultats complètement différents. Chaque opérateur et chaque fonction choisit le type de valeur qui semble le plus approprié de

retourner dans un contexte scalaire. Certains opérateurs retournent la longueur de la liste qui aurait été retournée dans un contexte de liste. D'autres opérateurs retournent la première valeur. D'autres encore retournent la dernière valeur. D'autres enfin retournent le nombre d'opérations réussies. En général, ils font ce que vous souhaitez, à moins que vous ne vouliez de la cohérence.

Un tableau nommé en contexte scalaire est assez différent de ce qui apparaîtrait au premier coup d'oeil comme une liste dans un contexte scalaire. Vous ne pouvez pas transformer une liste comme (1, 2, 3) dans un contexte scalaire, car le compilateur connaît le contexte à la compilation. Il générerait ici l'opérateur scalaire virgule, et non pas la version construction de liste de la virgule. Ce qui signifie que ça n'a jamais été considéré comme une liste avec laquelle travailler.

En général, les fonctions en Perl qui encapsulent les appels système du même nom (comme `chown(2)`, `fork(2)`, `close-dir(2)`, etc.) retournent toutes vrai quand elles réussissent et `undef` sinon, comme c'est souvent mentionné ci-dessous. C'est différent des interfaces C qui retournent -1 en cas d'erreur. Les exceptions à cette règle sont `wait`, `waitpid()` et `syscall()`. Les appels système positionnent aussi la variable spéciale `$!` en cas d'erreur. Les autres fonctions ne le font pas, sauf de manière accidentelle.

### 31.1.1 Fonctions Perl par catégories

Voici les fonctions Perl (y compris ce qui ressemble à des fonctions, comme certains mots-clés et les opérateurs nommés) triées par catégorie. Certaines fonctions apparaissent dans plusieurs catégories à la fois.

#### Fonctions liées aux scalaires ou aux chaînes de caractères

`chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q/CHAINE/`, `qq/CHAINE/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

#### Expressions rationnelles et reconnaissances de motifs

`m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`

#### Fonctions numériques

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

#### Fonctions liées aux véritables @tableaux

`pop`, `push`, `shift`, `splice`, `unshift`

#### Fonctions aux listes de données

`grep`, `join`, `map`, `qw/CHAINE/`, `reverse`, `sort`, `unpack`

#### Fonctions liées aux véritables %tables de hachage

`delete`, `each`, `exists`, `keys`, `values`

#### Fonctions d'entrée/sortie

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

#### Fonctions pour données de longueur fixe ou pour enregistrements

`pack`, `read`, `syscall`, `sysread`, `syswrite`, `unpack`, `vec`

#### Fonctions de descripteurs de fichiers, de fichiers ou de répertoires

`-X`, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

#### Mots-clés liés au contrôle d'exécution de votre programme Perl

`caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`, `wantarray`

#### Mots-clés liés à la portée

`caller`, `import`, `local`, `my`, `our`, `package`, `use`

#### Fonctions diverses

`defined`, `dump`, `eval`, `formline`, `local`, `my`, `our`, `reset`, `scalar`, `undef`, `wantarray`

#### Fonctions de gestion des processus et des groupes de processus

`alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx/CHAINE/`, `setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

#### Mots-clés liés aux modules perl

`do`, `import`, `no`, `package`, `require`, `use`

#### Mots-clés liés aux classes et à la programmation orientée objet

`bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

**Fonctions de bas niveau liées aux sockets**

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

**Fonctions IPC (communication inter-processus) System V**

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

**Manipulation des informations sur les utilisateurs et les groupes**

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

**Manipulation des informations du réseau**

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

**Fonction de date et heure**

gmtime, localtime, time, times

**Nouvelles fonctions de perl5**

abs, bless, chomp, chr, exists, formline, glob, import, lc, lcfirst, map, my, no, our, prototype, qx, qw, readline, readpipe, ref, sub\*, sysopen, tie, tied, uc, ucfirst, untie, use

\* - sub était un mot-clé dans perl4, mais dans perl5 c'est un opérateur qui peut être utilisé au sein d'expressions.

**Fonctions obsolètes en perl5**

dbmclose, dbmopen

**31.1.2 Portabilité**

Perl est né sur Unix et peut, par conséquent, accéder à tous les appels systèmes Unix courants. Dans des environnements non-Unix, les fonctionnalités de certains appels systèmes Unix peuvent manquer ou différer sur certains détails. Les fonctions Perl affectées par cela sont :

-X, binmode, chmod, chown, chroot, crypt, dbmclose, dbmopen, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, exec, fcntl, flock, fork, getgrent, getgrgid, gethostbyname, gethostent, getlogin, getnetbyaddr, getnetbyname, getnetent, getppid, getpgrp, getpriority, getprotobynumber, getprotoent, getpwent, getpwnam, getpwuid, getservbyport, getservent, getsockopt, glob, ioctl, kill, link, lstat, msgctl, msgget, msgrcv, msgsnd, open, pipe, readlink, rename, select, semctl, semget, semop, setgrent, sethostent, setnetent, setpgrp, setpriority, setprotoent, setpwent, setservent, setsockopt, shmctl, shmget, shmread, shmwrite, socket, socketpair, stat, symlink, syscall, sysopen, system, times, truncate, umask, unlink, utime, wait, waitpid

Pour de plus amples détails sur la portabilité de ces fonctions, voir *perlport* et toutes les documentations disponibles spécifiques à la plate-forme considérée.

**31.1.3 Fonctions Perl par ordre alphabétique****-X DESCRIPTEUR****-X EXPR****-X**

Un test de fichier où X est une des lettres présentées ci-dessous. Cet opérateur unaire prend soit un argument, soit un nom de fichier, soit un descripteur de fichier, et teste le fichier associé pour constater si quelque chose est vérifié à son sujet. Si l'argument est omis, il teste \$\_, sauf -t qui teste STDIN. Sauf indication contraire, il retourne 1 pour VRAI et " pour FAUX, ou la valeur indéfinie (undef) si le fichier n'existe pas. Malgré leurs noms originaux, leur priorité est la même que celle de tout autre opérateur unaire nommé et l'argument peut-être mis de même entre parenthèses. L'opérateur peut être :

- r Le fichier est en lecture par le uid/gid effectif.
- w Le fichier est en écriture par le uid/gid effectif.
- x Le fichier est exécutable par le uid/gid effectif.
- o Le fichier appartient au uid effectif.
- R Le fichier est en lecture par le uid/gid réel.
- W Le fichier est en écriture par le uid/gid réel.
- X Le fichier est exécutable par le uid/gid réel.
- O Le fichier appartient au uid réel.

- e Le fichier existe.
- z Le fichier a une taille nulle (il est vide).
- s Le fichier n'a pas une taille nulle (retourne sa taille en octets).
- f Le fichier est un fichier normal.
- d Le fichier est un répertoire.
- l Le fichier est un lien symbolique.
- p Le fichier est un tube nommé (FIFO), ou le descripteur est un pipe.
- S Le fichier est une socket.
- b Le fichier est un fichier blocs spécial.
- c Le fichier est un fichier caractères spécial.
- t Le fichier est ouvert sur un tty.
- u Le fichier a le bit setuid positionné.
- g Le fichier a le bit setgid positionné.
- k Le fichier a le sticky bit positionné.
- T Le fichier est un fichier texte ASCII (via une heuristique).
- B Le fichier est un fichier binaire (le contraire de -T).
- M La date de démarrage du script moins la date de dernière modification du fichier (exprimé en jours).
- A Idem pour le dernier accès au fichier.
- C Idem pour le dernier changement de l'inode du fichier (Unix peut avoir un comportement différent des autres systèmes).

Exemple :

```
while (<>) {
 chomp;
 next unless -f $_; # ignore les fichiers spéciaux
 #...
}
```

L'interprétation des opérateurs de permission sur le fichier `-r`, `-R`, `-w`, `-W`, `-x`, et `-X` est uniquement basée sur le mode du fichier et les uids/gids de l'utilisateur. En fait, il peut y avoir d'autres raisons pour lesquelles vous ne pouvez pas lire, écrire ou exécuter le fichier. Par exemple, le contrôle d'accès aux systèmes de fichiers réseau (NFS), les listes de contrôles d'accès (ACL), les systèmes de fichiers en lecture seule et les formats d'exécutable non reconnus.

Notez aussi que, pour le super-utilisateur, `-r`, `-R`, `-w`, et `-W` retournent toujours 1, et `-x` ainsi que `-X` retournent 1 si l'un des bits d'exécution est positionné dans le mode. Les scripts exécutés par le super-utilisateur peuvent donc nécessiter un appel `stat()` pour déterminer exactement les droits du fichier ou alors effectuer un changement temporaire d'uid.

Si vous utilisez les ACL (listes de contrôles d'accès), il existe un pragma appelé `filetest` qui peut produire des résultats plus précis que les informations minimales des bits de permissions fournies par `stat()`. Lorsque vous faites `use filetest 'access'`, les tests sur fichiers susnommés utiliseront les appels systèmes de la famille `access()`. Notez aussi que dans ce cas, les tests `-x` et `-X` peuvent retourner VRAI même si aucun bit d'exécution n'est positionné (que ce soit les bits normaux ou ceux des ACLs). Ce comportement étrange est dû à la définition des appels systèmes sous-jacents. Lisez la documentation du pragma `filetest` pour de plus amples informations. Notez que `-s/a/b/` n'effectue pas une substitution négative. Toutefois, écrire `-exp($foo)` fonctionne toujours comme prévu – seule une lettre isolée après un tiret est interprétée comme un test de fichier.

Les tests `-T` et `-B` fonctionnent de la manière suivante. Le premier bloc du fichier est examiné, à la recherche de caractères spéciaux tels que des codes de contrôle ou des octets avec un bit de poids fort. Si trop de caractères spéciaux (> 30 %) sont rencontrés, c'est un fichier `-B` sinon c'est un fichier `-T`. De plus, tout fichier contenant un octet nul dans le premier bloc est considéré comme binaire. Si `-T` ou `-B` est utilisé sur un descripteur de fichier, le tampon `stdio` courant est examiné à la place du premier bloc. `-T` et `-B` retournent tous les deux VRAI sur un fichier nul, ou une fin de fichier s'il s'agit d'un descripteur. Comme vous devez lire un fichier pour effectuer le test `-T`, la plupart du temps, vous devriez d'abord utiliser un `-f` sur le fichier, comme dans `next unless -f $file && -T $file`.

Si le descripteur spécial, constitué d'un seul underscore (N.d.T. : caractère souligné), est fourni comme argument d'un test de fichier (ou aux opérateurs `stat()` et `lstat()`), alors c'est la structure `stat` du dernier fichier traité par un test (ou opérateur) qui est utilisée, épargnant ainsi un appel système. (Ceci ne fonctionne pas avec `-t` et n'oubliez pas que `lstat()` et `-l` laisseront dans la structure `stat` des informations liées au fichier symbolique et non au fichier réel.) (Notez aussi que si des informations issues d'un appel à `lstat` étaient dans le buffer `stat` alors `-T` et `-B` les remplacent par le résultat de `stat _`.) Exemple :

```

print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;

```

**abs VALEUR****abs**

Retourne la valeur absolue de son argument. Si VALEUR est omis, utilise \$\_.

**accept NOUVELLESOCKET,GENERIQUE SOCKET**

Accepte une connexion entrante de socket, tout comme l'appel système `accept(2)`. Retourne l'adresse compacte en cas de succès, FAUX sinon. Voir l'exemple de `Sockets : communication client/serveur` in *perlipc*.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (`close-on-exec`) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de \$F. Voir \$F in *perlvar*.

**alarm SECONDES****alarm**

S'arrange pour qu'un SIGALRM soit délivré au processus après que le nombre spécifié de secondes s'est écoulé. Si SECONDES est omis, la valeur de \$\_ est utilisée. (Sur certaines machines, malheureusement, le temps écoulé peut être jusqu'à une seconde de plus ou de moins que celui spécifié, en fonction de la façon dont les secondes sont comptées. De plus, la partage du temps entre les différents processus peut entraîner un retard supplémentaire.)

Il n'est pas possible d'activer plusieurs décomptes temporels à la fois. Chaque appel annule le décompte précédent. La valeur 0 peut être fournie pour annuler le décompte précédent sans en créer un nouveau. La valeur retournée est le temps restant de décompte précédent.

Pour des délais d'une précision inférieure à la seconde, vous pouvez utiliser soit la version Perl à quatre paramètres de `select()` en laissant les trois premiers indéfinis soit l'interface `syscall()` de Perl pour accéder à `setitimer(2)` si votre système le supporte. Le module `Time::HiRes` (qui fait partie de Perl depuis la version 5.8 ou disponible sur CPAN sinon) peut aussi s'avérer utile.

C'est souvent une erreur de mélanger des appels à `alarm()` avec des appels à `sleep()` (`sleep` peut-être implémenté via des appels internes à `alarm` sur votre système.)

Si vous souhaitez utiliser `alarm()` pour contrôler la durée d'un appel système, il vous faut utiliser le couple `eval()/die()`. Vous ne pouvez pas compter sur l'alarme qui déclenche l'échec de l'appel système avec \$! positionné à `EINTR` car Perl met en place des descripteurs de signaux pour redémarrer ces appels sur certains systèmes. Utiliser `eval()/die()` fonctionne toujours sous réserve de l'avertissement signalé dans `Signaux` in *perlipc*.

```

eval {
 local $SIG{ALRM} = sub { die "alarm\n" }; # N.B. : \n obligatoire
 alarm $timeout;
 $nread = sysread SOCKET, $buffer, $size;
 alarm 0;
};
if ($?) {
 die unless $? eq "alarm\n"; # propage des erreurs inattendues
 # délai dépassé : time out
}
else {
 # délai non dépassé
}

```

Pour plus d'information, voir *perlipc*.

**atan2 Y,X**

Retourne l'arc-tangente de Y/X dans l'intervalle -PI à PI.

Pour l'opération tangente, vous pouvez utiliser la fonction POSIX : `tan()` ou la relation habituelle :

```

sub tan { sin($_[0]) / cos($_[0]) }

```

Notez que `atan2(0, 0)` n'est pas clairement défini.

### **bind SOCKET,NOM**

Associe une adresse réseau à une socket, tout comme l'appel système `bind`. Retourne VRAI en cas de succès, FAUX sinon. NOM doit être une adresse compactée (par `pack()`) du type approprié pour la socket. Voir les exemples de Sockets : communication client/serveur in *perlipc*.

### **binmode DESCRIPTEUR, FILTRE**

#### **binmode DESCRIPTEUR**

Positionne DESCRIPTEUR pour être lu ou écrit en mode "binaire" ou en mode "texte" sur les systèmes d'exploitation qui font la distinction entre fichiers texte et binaire. Si DESCRIPTEUR est une expression, sa valeur est utilisée comme nom du descripteur. Retourne vrai en cas de succès ou undef en cas d'échec en positionnant \$! (errno) dans ce dernier cas.

Sur certains systèmes (en général, les systèmes DOS ou Windows et dérivés), `binmode()` est nécessaire lorsque vous voulez travailler sur un fichier autre qu'un fichier texte. Dans un but de portabilité, il est donc conseillé de toujours utiliser `binmode()` lorsque c'est approprié et de ne jamais l'utiliser dans les autres cas. Cela permet aussi de gérer des entrées/sorties encodées par défaut en UTF-8 Unicode et non en octets.

En d'autres termes, quelle que soit la plate-forme, utilisez `binmode()` pour les données binaires, comme des images par exemple.

Si FILTRE est présent alors c'est une seule chaîne de caractères qui peut contenir plusieurs directives. Ces directives modifient le comportement du descripteur. Cela a un sens d'utiliser `binmode` sur un fichier texte lorsqu'on utilise FILTRE.

Si FILTRE est omis ou s'il vaut `:raw`, le DESCRIPTEUR est positionné pour passer des données binaires. Cela inclut la désactivation d'une éventuelle traduction des CRLF et passe en mode octets (à opposer à des caractères Unicode). Notez bien que, en dépit de ce qui est dit dans *"Programming Perl"* (le Camel book) ou ailleurs, `:raw` n'est pas l'inverse de `:crlf` – toute autre filtre (FILTRE ou LAYER en anglais) qui pourrait modifier la nature binaire du flot est aussi désactivée. Voir *PerlIO, perlrun* et tout ce qui est dit de la variable d'environnement PERLIO.

Les directives comme `:bytes`, `:crlf`, `:utf8` et plus généralement de la forme `...` sont appelées des filtres d'entrée/sortie. La directive `open` permet de choisir les filtres d'entrée/sortie utilisés par défaut. Voir *open*.

*Le paramètre FILTRE de la fonction binmode() est appelé "DISCIPLINE" dans "Programming Perl, 3rd Edition". Mais, depuis la publication de ce livre connu sous le nom de "Camel III", un consensus pour le nommage de ce paramètre est passé de "discipline" à "filtre" (layer en anglais). Toute la documentation de cette version de Perl se réfère donc maintenant à "filtre" ("layer") plutôt qu'à "discipline". Revenons maintenant à la documentation...*

Pour marquer un DESCRIPTEUR comme UTF-8, utilisez `:utf8`.

En général, `binmode()` devrait être appelé après `open()` et avant n'importe quelle opération d'entrée/sortie. L'appel à `binmode()` videra tous les tampons de données en attente de sortie (ou d'entrée). La seule exception est le filtre `:encoding` qui change l'encodage par défaut du DESCRIPTEUR. Voir *open*. L'appel au filtre `:encoding` est parfois nécessaire en cours d'usage d'un flot : il ne vide pas les tampons. Le filtre `:encoding` ajoute implicitement le filtre `:utf8` au-dessus de lui puisque Perl utilise en interne de caractères Unicode encodés en UTF-8.

Le système d'exploitation, les pilotes de périphériques, les bibliothèques C et l'interpréteur de Perl coopèrent afin de permettre au programmeur de considérer une fin de ligne comme un seul caractère (`\n`) et cela, indépendamment de sa représentation externe. Sur la plupart des systèmes d'exploitation, la représentation utilisée par les fichiers textes natifs correspond à la représentation interne mais sur certaines plates-formes la représentation externe de `\n` est constituée de plus d'un caractère.

Mac OS, toutes les variantes d'UNIX et les Stream\_LF de VMS utilisent un seul caractère pour représenter une fin de ligne dans leur représentation externe des textes (même si ce caractère unique est un RETOUR CHARIOT sur Mac OS et un SAUT DE LIGNE sur Unix et dans la plupart des fichiers VMS). Sur d'autres systèmes tels que VMS, MS-DOS et les différentes versions de MS-Windows, votre programme voit un `\n` comme un simple `\cJ` mais ce qui est réellement stocké dans les fichiers textes est le couple de caractères `\cM\cJ`. Cela signifie que, si vous n'utilisez pas `binmode()` sur ces systèmes, les séquences `\cM\cJ` sur disque seront converties en `\n` en entrée et que tous les `\n` produits par votre programme seront reconvertis en `\cM\cJ` à la sortie. C'est ce que vous voulez pour les fichiers textes mais cela peut être désastreux pour un fichier binaire.

Autre conséquence de l'utilisation de `binmode()` (sur certains systèmes) : les marqueurs spéciaux de fin de fichiers seront vus comme faisant partie du flux de données. Pour les systèmes de la famille Microsoft, cela signifie que, si vos données binaires contiennent un `\cZ`, le système d'entrée/sortie le considérera comme une fin de fichier à moins que vous n'utilisiez `binmode()`.

`binmode()` est important non seulement pour les opérations `readline()` et `print()` mais aussi lorsque vous utilisez `read()`, `seek()`, `sysread()`, `syswrite()` et `tell()` (voir *perlport* pour plus d'informations). Voir aussi `$/` et `$$` dans *perlvar* pour savoir comment fixer manuellement les séquences de fin de lignes en entrée et en sortie.

### **bless REF,NOMCLASSE**

**bless REF**

Cette fonction précise à la chose référencée par REF qu'elle est désormais un objet du package NOMCLASSE – ou le package courant si aucun NOMCLASSE n'est spécifié, ce qui est souvent le cas. Étant donné que l'appel à `bless()` est souvent la dernière instruction d'un constructeur, cette fonction retourne la référence elle-même. Utilisez toujours la version à deux arguments puisqu'une classe dérivée risque d'hériter de la fonction effectuant la "bénédiction" `bless()`. Cf. *perltoot* et *perlobj* pour de plus amples informations sur la notion de bénédiction d'objets.

Ne blessez des objets qu'avec des NOMCLASSES mélangeant des majuscules et des minuscules. Les espaces de nommages entièrement en minuscule sont réservés pour les directives (pragmas) Perl. Les types prédéfinis utilisent les noms entièrement en majuscule si bien que, pour éviter toute confusion, vous ne devez pas utiliser ces deux types de nommage. Soyez sûr que NOMCLASSE est une valeur vraie.

Voir Modules Perl in *perlmod*.

**caller EXPR****caller**

Retourne le contexte de l'appel de subroutine courant. Dans un contexte scalaire, retourne le nom du package de l'appelant, s'il existe, c'est-à-dire si nous sommes dans une subroutine, un `eval()` ou un `require()`, et retourne la valeur indéfinie (`undef`) sinon. En contexte de liste, retourne

```
($package, $filename, $line) = caller;
```

Avec EXPR, il retourne des informations supplémentaires que le débogueur utilise pour afficher un historique de la pile. La valeur de EXPR donne le nombre de contextes d'appels à examiner au-dessus de celui en cours.

```
($package, $filename, $line, $subroutine, $hasargs,
 $wantarray, $evaltext, $is_require, $hints, $bitmask) = caller($i);
```

Ici, `$subroutine` peut être `"(eval)"` si le cadre n'est pas un appel de routine mais un `eval()`. Dans un tel cas, les éléments supplémentaires `$evaltext` et `$is_require` sont positionnés : `$is_require` est vrai si le contexte est créé par un `require` ou un `use`, `$evaltext` contient le texte de l'instruction `eval EXPR`. En particulier, pour une instruction `eval BLOC`, `$filename` vaut `"(eval)"` mais `$evaltext` est indéfini. (Notez aussi que chaque instruction `use` crée un contexte `require` à l'intérieur d'un contexte `eval EXPR`.) `$subroutine` peut aussi être `(unknown)` si cette subroutine particulière a disparu de la table des symboles. `$hasargs` est vrai si une nouvelle instance de `@_` a été créé pour ce contexte. Les valeurs de `$hints` et de `$bitmask` risquent de changer d'une version de Perl à une autre. Elles n'ont donc aucun intérêt pour une utilisation externe.

De plus, s'il est appelé depuis le package DB, `caller` retourne plus de détails : il affecte à la liste de variables `@DB::args` les arguments avec lesquels la routine a été appelée.

Prenez garde à l'optimiseur qui a pu optimiser des contextes d'appel avant que `caller()` ait une chance de récupérer les informations. Ce qui signifie que `caller(N)` pourrait ne pas retourner les informations concernant le contexte d'appel que vous attendez, pour `N > 1`. En particulier, `@DB::args` peut contenir des informations relatives à un appel précédent de `caller()`.

**chdir EXPR****chdir DESCRIPTEUR (de fichier)****chdir DESCRIPTEUR (de répertoire)****chdir**

Change le répertoire courant à EXPR, si c'est possible. Si EXPR est omis, change vers le répertoire spécifié par la variable `$ENV{HOME}` si elle est définie ; sinon, c'est le répertoire spécifié par la variable `$ENV{LOGDIR}` qui est utilisé. (Sous VMS, la variable `$ENV{SYS$LOGIN}` est aussi regardée et utilisée si elle est définie.) Si aucune de ces variables n'est définie, `chdir` seul ne fait rien. Retourne VRAI en cas de succès, FAUX sinon. Cf. exemple de `die()`.

**chmod LISTE**

Change les permissions d'une liste de fichiers. Le premier élément de la liste doit être le mode numérique, qui devrait être un nombre en octal et *non* une chaîne de chiffres en octal : `0644` est correct, mais pas `'0644'`. Retourne le nombre de fichiers dont les permissions ont été changées avec succès. Voir aussi `oct`, si vous ne disposez que d'une chaîne de chiffres.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
$mode = '0644'; chmod $mode, 'foo'; # !!! fixe le mode à
 # --w---r-T
$mode = '0644'; chmod oct($mode), 'foo'; # ceci est mieux
$mode = 0644; chmod $mode, 'foo'; # cela est le meilleur
```



Sur les systèmes qui connaissent `chmod`, vous pouvez passer un descripteur de fichier à la place d'un nom de fichier. Sur les systèmes ne connaissant pas `chmod`, cela produira une erreur fatal lors de l'exécution.

Vous pouvez aussi importer les constantes symboliques `S_I*` du module `Fcntl` :

```
use Fcntl ':mode';

chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
Identique au chmod 0755 de l'exemple précédent.
```

### chomp VARIABLE

#### chomp( LISTE )

##### chomp

Cette version sûre de `chop` supprime toute fin de ligne correspondant à la valeur courante de `$/` (connue aussi sous le nom de `$INPUT_RECORD_SEPARATOR` dans le module `English`). Elle retourne le nombre total de caractères effacés de tous ses arguments. Elle est souvent utilisée pour effacer le saut de ligne de la fin d'une entrée quand vous vous souciez que l'enregistrement final pourrait ne pas avoir ce saut de ligne. En mode paragraphe (`$/ = ""`), elle efface tous les sauts de ligne à la fin de la chaîne de caractères. En mode «slurp» (`$/ = undef`) ou en mode enregistrement de taille fixe (`$/` est une référence vers un entier ou similaire, voir *perlvar*) `chomp()` ne supprime rien du tout. Si `VARIABLE` est omis, elle tronque `$_`. Exemple :

```
while (<>) {
 chomp; # évite le \n du dernier champ
 @array = split(/:/);
 # ...
}
```

Vous pouvez en fait tronquer tout ce qui est une lvalue, y compris les affectations :

```
chomp($cwd = 'pwd');
chomp($answer = <STDIN>);
```

Si vous tronquez une liste, chaque élément est tronqué et le nombre total de caractères effacés est retourné.

Si la directive `encoding` est active alors les longueurs retournées sont calculées en fonction de la longueur de `$/` exprimée en nombre de caractères Unicode, qui n'est pas toujours la même si elle est exprimée en fonction de l'encodage natif.

Notez que les parenthèses sont nécessaires lorsque vous voulez "chomp"-er quelque chose qui n'est pas une simple variable. C'est parce que `chomp $cwd = 'pwd'` est interprété comme `(chomp $cwd) = 'pwd'` ; plutôt que comme `chomp( $cwd = 'pwd' )` ;. De même dans le cas de `chomp $a, $b` qui est interprété comme `chomp($a), $b` plutôt que comme `chomp($a, $b)`.

### chop VARIABLE

#### chop( LISTE )

##### chop

Efface le dernier caractère d'une chaîne et le retourne. Cet opérateur est utilisé pour effacer le saut de ligne de la fin d'une entrée car il est plus efficace que `s/\n//` étant donné qu'il ne scanne ni ne copie la chaîne. Si `VARIABLE` est omis, tronque `$_`. Exemple :

```
while (<>) {
 chop; # évite \n du dernier champ
 @array = split(/:/);
 #...
}
```

Vous pouvez en fait tronquer tout ce qui est une lvalue, y compris les affectations :

```
chop($cwd = 'pwd');
chop($answer = <STDIN>);
```

Si vous tronquez une liste, chaque élément est tronqué. Seul la valeur du dernier `chop()` est retournée.

Notez que `chop()` retourne le dernier caractère. Pour retourner tout sauf le dernier caractère, utilisez `substr($string, 0, -1)`.

Voir aussi `chomp`.

**chown LISTE**

Change le propriétaire (et le groupe) d'une liste de fichiers. Les deux premiers éléments de la liste doivent être, dans l'ordre, les uid et gid *numériques*. Une valeur -1 à l'une de ces positions est interprétée par la plupart des systèmes comme une valeur à ne pas modifier. Retourne le nombre de fichiers modifiés avec succès.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Sur les systèmes qui connaissent fchown, vous pouvez utiliser des descripteurs de fichier à la place des noms de fichier. Sur les systèmes ne connaissant pas fchown, cela produira une erreur fatal lors de l'exécution.

Voici un exemple qui cherche les uid non numériques dans le fichier de mots de passe :

```
print "User: ";
chop($user = <STDIN>);
print "Files: ";
chop($pattern = <STDIN>);

($login,$pass,$uid,$gid) = getpwnam($user)
 or die "$user not in passwd file";

@ary = glob($pattern); # expansion des noms de fichiers
chown $uid, $gid, @ary;
```

Sur la plupart des systèmes, vous n'êtes pas autorisé à changer le propriétaire d'un fichier à moins d'être le super-utilisateur, même si avez la possibilité de changer un groupe en l'un de vos groupes secondaires. Sur les systèmes non sécurisés, ces restrictions peuvent être moindres, mais ceci n'est pas une hypothèse portable. Sur les systèmes POSIX, vous pouvez détecter cette condition de la manière suivante :

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
$can_chown_giveaway = not sysconf(_PC_CHOWN_RESTRICTED);
```

**chr NOMBRE****chr**

Retourne le caractère représenté par ce NOMBRE dans le jeu de caractères. Par exemple, chr(65) est "A" en ASCII ou Unicode et chr(0x263a) est un visage réjoui en Unicode. Notez que les caractères de 128 à 255 (inclu) ne sont pas par défaut encodés en UTF-8 Unicode pour des raisons de compatibilités (mais voir *encoding*).

Si NOMBRE est omis, s'applique à \$\_.

Pour la fonction réciproque, utilisez ord.

Notez que si la directive bytes est active, NOMBRE est masqué pour ne conserver que les 8 bits de poids faible.

Voir *perlunicode* et *encoding* pour en savoir plus sur Unicode.

**chroot FICHER****chroot**

Cet opérateur fonctionne comme l'appel système du même nom : il fait du répertoire spécifié le nouveau répertoire racine pour tous les chemins commençant par un "/" utilisés par votre processus et ses enfants. (Ceci n'affecte pas le répertoire courant qui reste inchangé.) Pour des raisons de sécurité, cet appel est restreint au super-utilisateur. Si FICHER est omis, effectue un chroot() sur \$\_.

**close DESCRIPTEUR****close**

Ferme le fichier ou le tube associé au descripteur de fichier, en retournant VRAI si et seulement si les tampons d'entrée/sortie ont été correctement vidés et si le descripteur a été correctement fermé. Ferme le descripteur courant si l'argument est omis.

Vous n'avez pas à fermer le DESCRIPTEUR si vous allez immédiatement refaire un open() sur celui-ci, car open() le fermera pour vous. (voir open().) Toutefois, un close() explicite sur un fichier d'entrée réinitialise le compteur de lignes (\$) alors que la fermeture implicite par un open() ne le fait pas.

Si le descripteur vient d'un tube ouvert, close() va de plus retourner FAUX si un des autres appels système impliqués échoue ou si le programme se termine avec un statut non nul. (Si le seul problème rencontré est une terminaison de programme non nulle, \$! sera à 0.) De même, la fermeture d'un tube attend que le programme exécuté sur le tube soit achevé, au cas où vous souhaiteriez voir la sortie du tube après coup, et positionne implicitement la valeur du statut de sortie de la commande dans \$?.

La fermeture prématurée d'une extrémité de lecture d'un tube (c'est-à-dire avant que le processus qui écrit à l'autre extrémité l'ait fermé) aura pour conséquence l'envoi d'un signal SIGPIPE au processus écrivain. Si l'autre extrémité n'est pas prévue pour gérer cela, soyez sûr d'avoir lu toutes les données avant de fermer le tube.

Exemple :

```

open(OUTPUT, '|sort >foo') # tube vers sort
 or die "Can't start sort: $!";
#... # imprime des trucs sur la sortie
close OUTPUT # attend la fin de sort
 or warn $! ? "Error closing sort pipe: $!"
 : "Exit status $? from sort";
open(INPUT, 'foo') # recupere les resultats de sort
 or die "Can't open 'foo' for input: $!";

```

Le DESCRIPTEUR peut être une expression dont la valeur peut être utilisée en tant que descripteur indirect, habituellement le véritable nom du descripteur.

### closedir REPDESCRIPTEUR

Ferme un répertoire ouvert par `opendir()` et retourne le résultat de cet appel système.

### connect SOCKET,NOM

Tente une connexion sur une socket distante, tout comme le fait l'appel système du même nom. Retourne VRAI en cas de succès, FAUX sinon. Le NOM doit être une adresse compactée (par `pack()`) du type approprié correspondant à la socket. Voir les exemples de Sockets : communication Client/Serveur in *perlipc*.

### continue BLOC

En fait, `continue` est une instruction de contrôle d'exécution plutôt qu'une fonction. S'il y a un BLOC `continue` rattaché à un BLOC (typiquement dans un `while` ou un `foreach`), il est toujours exécuté juste avant le test à évaluer à nouveau, tout comme la troisième partie d'une boucle `for` en C. Il peut donc être utilisé pour incrémenter une variable de boucle, même si la boucle a été continuée par l'instruction `next` (qui est similaire à l'instruction `continue` en C).

`last`, `next`, ou `redo` peuvent apparaître dans un bloc `continue`. `last` et `redo` vont se comporter comme s'ils avaient été exécutés dans le bloc principal. De même pour `next`, mais comme il va exécuter un bloc `continue`, il peut s'avérer encore plus divertissant.

```

while (EXPR) {
 ### redo vient toujours ici
 do_something;
} continue {
 ### next vient toujours ici
 do_something_else;
 # puis retour au sommet pour révérifier EXPR
}
last vient toujours ici

```

Omettre la section `continue` est sémantiquement équivalent à en utiliser une vide, de manière assez logique. Dans ce cas, `next` retourne directement vérifier la condition au sommet de la boucle.

### cos EXPR

Retourne le cosinus de EXPR (exprimé en radians). Si EXPR est omis, calcule le cosinus de `$_`.

Pour la fonction réciproque (arc cosinus), vous pouvez utiliser la fonction `Math::Trig::acos()` ou alors utiliser cette relation :

```

sub acos { atan2(sqrt(1 - $_[0] * $_[0]), $_[0]) }

```

### crypt TEXTE,SEL

Crée une empreinte exactement comme le fait la fonction `crypt(3)` de la bibliothèque C (en supposant que vous n'avez pas une version dégradée car considéré comme arme potentielle).

`crypt()` est une fonction non réversible (à sens unique). Le TEXTE et le SEL sont transformés en une courte chaîne de caractères, appelée empreinte, qui est alors retournée. Un même couple TEXTE, SEL donne toujours la même empreinte mais on ne connaît aucun moyen de retrouver le TEXTE à partir de l'empreinte. Un petit changement dans TEXTE ou SEL donne un résultat complètement différent pour l'empreinte.

Il n'existe pas de fonction `decrypt`. La fonction `crypt()` n'est donc pas utilisable pour chiffrer des documents (pour cela, chercher les modules *crypt* sur CPAN) et donc le nom "crypt" n'est pas vraiment bien choisi. En fait, elle sert plutôt à vérifier que deux textes sont identiques sans nécessiter l'envoi ou le stockage du texte lui-même. On peut par exemple vérifier un mot de passe. Seule l'empreinte du mot de passe est stockée. L'utilisateur tape son mot de passe et `crypt()` calcule son empreinte avec le même SEL qui celui utilisé pour stocker le mot de passe. Si les deux empreintes (celle stockée et celle calculée) correspondes alors le mot de passe est correct.

Pour vérifier une empreinte, vous devriez utiliser cette empreinte comme SEL (par exemple `crypt($motdepasse, $empreinte) eq $empreinte`). Le SEL utilisé pour créer l'empreinte apparaît en clair dans l'empreinte. Cela

permet de garantir que `crypt()` hachera le nouveau `TEXT` avec le même `SEL` que celui de l’empreinte. Cela vous garantit aussi que votre code fonctionnera avec une version standard de `crypt` mais aussi avec des implémentations plus exotiques. En d’autres termes, ne faites aucune supposition sur la forme de l’empreinte produite ou sur sa taille. Traditionnellement, le résultat est une chaîne de 13 octets : les deux premiers octets du `SEL`, suivis de 11 octets pris dans l’ensemble `[./0-9A-Za-z]` et seuls les huit premiers octets de la chaîne à encrypter sont pris en compte. Mais des méthodes de hachages alternatives (comme MD5) ou d’un niveau de sécurité plus élevé (comme C2) ou encore des implémentations sur des plateformes non-UNIX peuvent produire d’autres types de chaînes.

Lorsque vous choisissez un nouveau `SEL` constitué de deux caractères aléatoires, ces caractères doivent provenir de l’ensemble `[./0-9A-Za-z]` (Exemple `join "", ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`). Cet ensemble de caractères n’est qu’une recommandation car les caractères réellement acceptables dans `SEL` dépendent de la fonction `crypt` de votre bibliothèque système et Perl ne peut pas connaître les restrictions exactes.

Voici un exemple qui garantit que quiconque lance ce programme connaît son propre mot de passe :

```
$pwd = (getpwuid($<))[1];
$salt = substr($pwd, 0, 2);

system "stty -echo";
print "Password: ";
chop($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $salt) ne $pwd) {
 die "Sorry...\n";
} else {
 print "ok\n";
}
```

Bien évidemment, donner votre mot de passe à quiconque vous le demande est très peu recommandé.

La fonction `crypt` n’est pas utilisable pour chiffrer de grande quantité d’information déjà parce que vous ne pouvez pas retrouver l’information initiale. Regarder le module *Digest* pour des algorithmes plus robustes.

Si vous utilisez `crypt()` sur une chaîne Unicode (qui contient *éventuellement* des caractères dont l’encodage est supérieur à 255), Perl essaie de se sortir de cette situation en dégradant une copie de la chaîne vers un encodage sur 8 bits avant d’appeler `crypt()` sur cette copie. Si cela fonctionne, tout est bon. Sinon, `crypt()` meurt (`die`) avec les message `Wide character in crypt`.

### **dbmclose HASH**

[Cette fonction a été avantageusement remplacée par la fonction `untie`.]

Rompt le lien entre un fichier DBM et une table de hachage.

### **dbmopen HASH,DBNOM,MODE**

[Cette fonction a été avantageusement remplacée par la fonction `tie()`.]

Cette fonction lie un fichier `dbm(3)`, `ndbm(3)`, `sdbm(3)`, `gdbm(3)`, ou Berkeley DB à une table de hachage. `HASH` est le nom de la table de hachage. (À la différence du `open()` normal, le premier argument n’est *pas* un descripteur de fichier, même s’il en a l’air). `DBNOM` est le nom de la base de données (sans l’extension `.dir` ou `.pag`, le cas échéant). Si la base de données n’existe pas, elle est créée avec les droits spécifiés par `MODE` (et modifiés par `umask`). Si votre système supporte uniquement les anciennes fonctions DBM, vous ne pouvez exécuter qu’un seul `dbmopen()` dans votre programme. Dans les anciennes versions de Perl, si votre système n’avait ni DBM ni `ndbm`, l’appel à `dbmopen()` produisait une erreur fatale ; il utilise maintenant `sdbm(3)`.

Si vous n’avez pas les droits d’écriture sur le fichier DBM, vous pouvez seulement lire les variables de la table de hachage, vous ne pouvez pas y écrire. Si vous souhaitez tester si vous pouvez y écrire, faites un test sur le fichier ou essayez d’écrire une entrée bidon dans la table de hachage, à l’intérieur d’un `eval()`, qui interceptera l’erreur.

Notez que les fonctions telles que `keys()` et `values()` peuvent retourner des listes gigantesques si elles sont utilisées sur de gros fichiers DBM. Vous devriez préférer la fonction `each()` pour parcourir des fichiers DBM volumineux. Exemple :

```
imprime en sortie les index du fichier d'historiques
dbmopen(%HIST, '/usr/lib/news/history', 0666);
while (($key,$val) = each %HIST) {
 print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

Voir aussi `AnyDBM_File` pour une description plus générale des avantages et inconvénients des différentes approches dbm ainsi que `DB_File` pour une implémentation particulièrement riche.

Vous pouvez contrôler la bibliothèque DBM utilisé en chargeant cette bibliothèque avant l'appel à `dbmopen()` :

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db")
 or die "Can't open netscape history file: $!";
```

### defined EXPR

#### defined

Retourne une valeur booléenne exprimant si EXPR a une valeur autre que la valeur indéfinie `undef`. Si EXPR est absent, `$_` sera vérifiée de la sorte.

De nombreuses opérations retournent `undef` pour signaler un échec, la fin d'un fichier, une erreur système, une variable non initialisée et d'autres conditions exceptionnelles. Cette fonction vous permet de distinguer `undef` des autres valeurs. (Un simple test booléen ne distinguera pas `undef`, zéro, la chaîne de caractères vide et `<"0">`, qui représentent tous faux.) Notez que puisque `undef` est un scalaire valide, sa présence n'indique pas *nécessairement* une condition exceptionnelle : `pop()` retourne `undef` quand son argument est un tableau vide *ou* quand l'élément à retourner a la valeur `undef`.

Vous pouvez aussi utiliser `defined(&func)` pour vérifier si la subroutine `&func` a déjà été définie. La valeur retournée ne sera pas affectée par une déclaration préalable de `&func`. Notez qu'une subroutine même non définie peut malgré tout être appellable : son package a peut-être une méthode `AUTOLOAD` qui la définira lors de son premier appel – voir *perlsub*.

L'utilisation de `defined()` sur des agrégats (tables de hachage et tableaux) est dépréciée. C'était utilisé pour savoir si de la mémoire avait déjà été allouée pour cet agrégat. Ce comportement pourrait disparaître dans une future version de Perl. Vous devriez utiliser un simple test de taille à la place :

```
if (@an_array) { print "has array elements\n" }
if (%a_hash) { print "has hash members\n" }
```

Utilisé sur un élément de table de hachage, il vous indique si la valeur est définie, mais pas si la clé existe dans la table. Utilisez `exists` dans ce but.

Exemples :

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
 unless defined($value = readlink $sym);
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
$dbugging = 0 unless defined $debugging;
```

Note : de nombreuses personnes ont tendance à trop utiliser `defined()` et sont donc surprises de découvrir que le nombre `0` et `""` (la chaîne de caractères vide) sont, en fait, des valeurs définies. Par exemple, si vous écrivez :

```
"ab" =~ /a(.*?)b/;
```

La recherche de motif réussit et `$1` est définie, bien qu'il ne corresponde à "rien". En fait, elle n'a pas véritablement rien trouvé – elle a plutôt trouvé quelque chose qui s'est avéré être d'une longueur de `0` caractères. Tout ceci reste très loyal et honnête. Quand une fonction retourne une valeur indéfinie, il est admis qu'elle ne pourrait pas vous donner une réponse honnête. Vous devriez donc utiliser `defined()` uniquement lorsque vous vous posez des questions sur l'intégrité de ce que vous tentez de faire. Sinon, une simple comparaison avec `0` ou `""` est ce que vous voulez.

Voir aussi `undef`, `exists`, `ref`.

### delete EXPR

À partir d'une expression EXPR qui spécifie un élément ou un ensemble d'éléments (slice) d'une table de hachage ou d'un tableau, supprime le(s) élément(s) spécifié(s) de la table de hachage ou du tableau. Dans le cas d'un tableau, si les éléments supprimés se trouvent à la fin du tableau, la taille du tableau est réduite à l'indice le plus grand de tous les indices des éléments restants qui donnent une valeur vraie lorsqu'on teste leur existence via `exists()` (ou `0` si aucun élément n'existe).

Retourne une liste contenant autant d'éléments que le nombre d'éléments pour lequel une suppression a été tenté. Chaque élément de cette liste est soit la valeur de l'élément supprimé soit la valeur indéfinie (`undef`). Dans un contexte scalaire, vous obtiendrez le dernier élément supprimé (ou la valeur indéfinie si cet élément n'existait pas).

```
%hash = (foo => 11, bar => 22, baz => 33);
$scalar = delete $hash{foo}; # $scalar vaut 11
$scalar = delete @hash{qw(foo bar)}; # $scalar vaut 22
@array = delete @hash{qw(foo bar baz)}; # @array vaut (undef,undef,33)
```

Supprimer des éléments dans %ENV modifie les variables d'environnement. Supprimer des éléments d'une table de hachage liée à un fichier DBM supprime ces éléments du fichier. Supprimer des éléments d'une table de hachage ou d'un tableau lié ne retourne pas nécessairement quelque chose.

Supprimer un élément d'un tableau réinitialise effectivement cet emplacement à sa valeur indéfinie initiale. Par conséquent, un test d'existence sur cet élément via exists() retournera faux. Par ailleurs, supprimer des éléments au milieu d'un tableau ne décale pas vers le bas les indices des éléments suivants. Utiliser splice() pour faire cela. Voir exists.

Le code suivant supprime (de manière inefficace) toutes les valeurs de %HASH et de @TABLEAU :

```
foreach $key (keys %HASH) {
 delete $HASH{$key};
}

foreach $index (0 .. $#TABLEAU) {
 delete $TABLEAU[$index];
}
```

De même que le code suivant :

```
delete @HASH{keys %HASH}

delete @TABLEAU[0 .. $#TABLEAU];
```

Ces deux méthodes restent toutefois beaucoup plus lentes que la simple assignation de la liste vide ou l'utilisation de undef() sur %HASH ou @TABLEAU :

```
%HASH = (); # vider complètement %HASH
undef %HASH; # oublier que %HASH a existé

@TABLEAU = (); # vider complètement @TABLEAU
undef @TABLEAU; # oublier que @TABLEAU a existé
```

Notez que EXPR peut être arbitrairement compliquée tant que l'opération finale se réfère à un élément ou une partie d'une table de hachage ou d'un tableau :

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};

delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

## die LISTE

En dehors d'un eval(), affiche les valeur de la LISTE sur STDERR et quitte avec la valeur actuelle de \$! (numéro d'erreur errno). Si \$! est 0, sort avec la valeur (\$? >> 8) (statut d'une 'commande' entre simples apostrophes inverses). Si (\$? >> 8) est 0, sort avec 255. À l'intérieur d'un eval(), le message d'erreur est mis dans \$@ et le eval() s'achève sur la valeur indéfinie. Ce qui fait de die() la façon de soulever une exception.

Exemples équivalents :

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

Si la dernière valeur de LISTE ne se termine pas par un saut de ligne, le numéro de la ligne actuelle du script et le numéro de la ligne d'entrée (le cas échéant) sont aussi affichés et un saut de ligne est ajouté. Notez que "le numéro de la ligne d'entrée" (mieux connu sous le nom de "chunk") est dépendant de la notion de "ligne" courante et est disponible dans la variable spéciale \$.. Voir \$/ in perlvar et \$. in perlvar.

Astuce : parfois, la concaténation de ", stopped" à votre message le rendra plus sensé lorsque la chaîne de caractères "at foo line 123" sera ajoutée. Supposez que vous exécutiez le script "canasta".

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

va respectivement produire

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

Voir aussi `exit()` et `warn()` et le module `Carp`.

Si la LISTE est vide et si `$@` contient déjà une valeur (provenant par exemple d'une évaluation précédente), cette valeur est réutilisée après la concaténation de `"\t...propagated"`. Ceci est utile pour propager des exceptions :

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

Si LISTE est vide et si `$@` contient une référence vers un objet qui a une méthode `PROPAGATE`, cette méthode sera appelée avec comme argument le nom du fichier et le numéro de ligne. La valeur retournée remplacera la valeur dans `$@` (comme si `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }` ; était appelé).

Si `$@` est vide, alors la chaîne de caractères "Died" est utilisée.

L'argument de `die()` peut aussi être une référence. Lorsque cela arrive à l'intérieur d'un `eval()` alors `$@` contiendra cette référence. Ce comportement autorise une implémentation plus élaborée de la gestion des exceptions utilisant des objets contenant une description arbitraire de la nature de l'exception. Une telle utilisation est parfois préférable à la reconnaissance d'un motif particulier par une expression rationnelle appliquée la valeur de `$@`. Voici un exemple :

```
use Scalar::Util 'blessed';

eval { ... ; die Some::Module::Exception->new(FOO => "bar") };
if ($@) {
 if (blessed($@) && $@->isa("Some::Module::Exception")) {
 # gestion de Some::Module::Exception
 }
 else {
 # gestion de toutes les autres exceptions
 }
}
```

Étant donné que perl transformera en chaîne toutes les exceptions non captées avant de les afficher, vous voudrez peut-être surcharger l'opération de transformation en chaîne de tous vos objets représentant des exceptions. Regardez *overload* pour faire cela.

Vous pouvez vous arranger pour qu'une subroutine (de callback) soit appelée juste après le `die()` en l'attachant à `$_SIG{__DIE__}`. La subroutine associée sera appelée avec le texte de l'erreur et peut changer le message de l'erreur, le cas échéant, en appelant `die()` à nouveau. Voir `$_SIG{expr}` in *perlvar* pour les détails sur l'assignation des entrées de `%SIG` et `eval BLOC ($??)` pour des exemples. Bien que cette fonctionnalité ait été conçue afin d'être utilisée juste au moment où votre script se termine, ce n'est pas le cas – la subroutine attachée à `$_SIG{__DIE__}` peut aussi être appelée lors de l'évaluation d'un bloc ou d'une chaîne ! Si vous voulez que votre subroutine ne fasse rien dans ce cas, utilisez :

```
die @_ if $^S;
```

comme première ligne de votre subroutine (Voir `$^S` in *perlvar*). Étant donné que cela peut engendrer des choses étranges, ce comportement contre-intuitif devrait être changé dans une prochaine version.

#### do BLOC

Pas vraiment une fonction. Retourne la valeur de la dernière commande dans la suite de commandes contenues dans BLOC. Lorsque suivi par l'une des commandes de boucle `while` ou `until`, exécute le BLOC une fois avant le test de la condition de boucle. (Dans les autres cas, les structures de boucle testent la condition d'abord.)

`do BLOC` n'est pas considéré comme une boucle et donc les instructions de contrôle de boucle `next`, `last` et `redo` ne peuvent pas être utilisées pour quitter ou redémarrer le bloc. Voir *perlsyn* pour des stratégies alternatives.

#### do ROUTINE(LISTE)

Forme dépréciée d'appel de routine. Voir *perlsub*.

#### do EXPR

Utilise la valeur de EXPR comme nom de fichier et exécute le contenu de ce fichier en tant que script Perl.

```
do 'stat.pl';
```

est identique à

```
eval 'cat stat.pl';
```

sauf que c'est plus efficace et concis, qu'une trace du fichier courant est gardée pour les messages d'erreur, que la recherche du fichier a lieu dans tous les répertoires de @INC et que, si le fichier est trouvé, %INC est mis à jour. Voir Noms prédéfinis in *perlvar* pour ces variables. De plus, les variables lexicales visibles lors de l'appel ne sont pas vues par `do` NOMFICHER alors qu'elle le sont par `eval` CHAINE. En revanche, dans les deux cas, le fichier est retraité à chaque fois que vous l'appeler ce qui n'est probablement pas ce que vous voudriez faire à l'intérieur d'une boucle.

Si `do` ne peut pas lire le fichier, il retourne `undef` et assigne l'erreur à `$!`. Si `do` peut lire le fichier mais non le compiler, il retourne `undef` et assigne un message d'erreur à `$@`. Si le fichier est compilé avec succès, `do` retourne la valeur de la dernière expression évaluée.

Notez que l'inclusion de modules de bibliothèques est mieux faite par les opérateurs `use()` et `require()` qui effectuent aussi une vérification automatique des erreurs et soulèvent une exception s'il y a le moindre problème.

Vous pourriez aimer utiliser `do` pour lire le fichier de configuration d'un programme. La vérification manuelle d'erreurs peut être effectuée de la façon suivante :

```
lecture des fichiers : d'abord le système puis l'utilisateur
for $file ("/share/prog/default.rc",
 "$ENV{HOME}/.someprogrc") {
 unless ($return = do $file) {
 warn "couldn't parse $file: $@" if $@;
 warn "couldn't do $file: $!" unless defined $return;
 warn "couldn't run $file" unless $return;
 }
}
```

## dump LABEL

### dump

Ceci provoque immédiatement la création de l'image mémoire du programme (core dump). Voir aussi l'option de ligne de commande `-u` dans *perlrun* qui fait la même chose. Au départ, ceci permet d'utiliser le programme **undump** (non fourni) pour convertir votre image mémoire en un programme binaire exécutable après avoir initialisé toutes vos variables au début du programme. Quand le nouveau binaire est exécuté, il va commencer par exécuter un `goto LABEL` (avec toutes les restrictions dont souffre `goto`). Pensez-y comme un branchement `goto` avec l'intervention d'une image mémoire et une réincarnation. Si LABEL est omis, relance le programme au début.

ATTENTION : tout fichier ouvert au moment de la copie ne sera PAS ouvert à nouveau quand le programme sera réincarné, avec pour résultat une confusion possible sur la portion de Perl.

Cet opérateur est très largement obsolète, en partie parce qu'il est très difficile de convertir un fichier d'image mémoire (core) en exécutable mais aussi parce que le véritable compilateur perl-en-C l'a surpassé. C'est pourquoi vous devez maintenant l'appeler par `CORE::dump()` si vous ne voulez pas recevoir un message d'avertissement au sujet d'une erreur de frappe possible.

Si vous projetez d'utiliser *dump* pour augmenter la vitesse de votre programme, essayez donc de produire du pseudo-code (ou bytecode) ou du C natif comme cela est décrit dans *perlcc*. Si vous essayez juste d'accélérer un script CGI, regardez donc du côté de l'extension `mod_perl` de **Apache** ou de celui du module `CGI::Fast` sur CPAN. Vous pouvez aussi envisager l'autochargement (autoloading ou selfloading) qui donnera au moins la *sensation* que votre programme va plus vite.

## each HASH

Appelé dans un contexte de liste, retourne une liste de deux éléments constituée de la clé et de la valeur du prochain élément d'une table de hachage, de sorte que vous puissiez itérer sur celle-ci. Appelé dans un contexte scalaire, retourne uniquement la clé de l'élément suivant de la table de hachage.

Les entrées sont retournées dans un ordre apparemment aléatoire. Cet ordre aléatoire réel pourrait changer dans les versions futures de perl mais vous avez la garantie que cet ordre reste le même, que vous utilisiez la fonction `keys` ou la fonction `values` sur une même table de hachage (non modifiée). Depuis la version 5.8.1 de Perl, pour des raisons de sécurité, cet ordre change même à chaque exécution de Perl (voir *Attaques par complexité algorithmique* in *perlsec*).

Quand la table de hachage est entièrement lue, un tableau nul est retourné dans un contexte de liste (qui, lorsqu'il est assigné, produit une valeur FAUSSE 0), et `undef` dans un contexte scalaire. Le prochain appel à `each` après cela redémarrera une nouvelle itération. Il y a un seul itérateur pour chaque table de hachage, partagé par tous les appels à `each`, `keys` ou `values` du programme ; il peut être réinitialisé en lisant tous les éléments de la table ou en évaluant `keys HASH` ou `values HASH`. Si vous ajoutez ou supprimez des éléments d'une table de hachage pendant que vous itérez sur celle-ci, vous pourriez avoir des entrées ignorées ou dupliquées, donc ne le faites pas.

Le code suivant affiche votre environnement comme le fait le programme `printenv(1)`, mais dans un ordre différent :



```
while (($key,$value) = each %ENV) {
 print "$key=$value\n";
}
```

Voir aussi `keys()` et `values()`.

## eof DESCRIPTEUR

### eof ()

#### eof

Retourne 1 si la prochaine lecture sur un DESCRIPTEUR de fichier va retourner une fin de fichier, ou si le DESCRIPTEUR n'est pas ouvert. Le DESCRIPTEUR peut être une expression dont la valeur donne un véritable descripteur de fichier. (Notez que cette fonction lit en fait un caractère puis le retire comme `ungetc()`, elle n'est donc pas vraiment utile dans un contexte interactif.) Ne faites pas de lecture sur un fichier d'un terminal (ou d'appel à `eof(DESCRIPTEUR)` sur celui-ci) après qu'une fin de fichier soit atteinte. Les types de fichiers comme les terminaux peuvent perdre la condition de fin de fichier si vous le faites.

Un `eof` sans un argument utilise le dernier fichier lu comme argument. Utiliser `eof()` avec des parenthèses vides est très différent. Il se réfère alors au pseudo fichier formé par les fichiers listés sur la ligne de commande et lu par l'opérateur `<>`. Puisque `<>` n'est pas explicitement ouvert comme l'est un descripteur de fichier normal, l'utilisation de `eof()` avant celle de `<>` déclenchera l'examen de `@ARGV` pour voir si une entrée est disponible. De manière similaire, un `eof()` après que `<>` a retourné la condition fin-de-fichier supposera que vous traitez un autre fichier de la liste `@ARGV` ou que vous lisez depuis `STDIN`. Voir Les opérateurs d'E/S in *perlop*.

Dans une boucle `while (<>)` `eof(ARGV)` ou `eof` peuvent être utilisés pour tester la fin de *CHAQUE* fichier alors que `eof()` ne détectera que la fin du tout dernier fichier. Exemple :

```
réinitialise le numérotage des lignes sur chaque fichier d'entrée
while (<>) {
 next if /^\\s*#/; # saute les commentaires
 print "$.\t$_";
} continue {
 close ARGV if eof; # ce n'est pas eof() !
}

insère des tirets juste avant la dernière ligne du dernier fichier
while (<>) {
 if (eof()) { # vérifie la fin du dernier fichier
 print "-----\n";
 }
 print;
 last if eof(); # nécessaire si lecture depuis un terminal
}
```

Astuce pratique : vous n'avez presque jamais besoin d'utiliser `eof` en Perl parce que les opérateurs d'entrée retournent la valeur `undef` quand ils n'ont plus d'informations à lire ou s'il y a eu une erreur.

## eval EXPR

### eval BLOC

#### eval

Dans sa première forme, la valeur retournée par `EXPR` est parcourue puis exécutée comme un petit programme Perl. La valeur de l'expression (qui est elle-même déterminée dans un contexte scalaire) est d'abord parcourue puis, s'il n'y a pas eu d'erreurs, exécutée dans le contexte du programme courant, de telle sorte que toute assignation de variable, et toute définition de subroutine ou de format perdure après cette exécution. Notez que la valeur est parcourue à chaque exécution de `eval`. Si `EXPR` est omis, évalue `$_`. Cette forme est typiquement utilisée pour repousser la compilation et l'exécution du texte contenu dans `EXPR` jusqu'à l'exécution du programme.

Dans sa seconde forme, le code à l'intérieur du BLOC est parcouru une seule fois – au même moment que la compilation du code entourant le `eval` – et exécuté dans le contexte du programme Perl courant. Cette forme est typiquement utilisée pour intercepter des exceptions plus efficacement que la première (voir ci-dessous), en fournissant aussi le bénéfice d'une vérification du code dans le BLOC lors de la compilation.

Le point-virgule final, le cas échéant, peut être omis dans `EXPR` ou à l'intérieur du BLOC.

Dans les deux formes, la valeur retournée est la valeur de la dernière expression évaluée à l'intérieur du mini-programme ; il est aussi possible d'utiliser un retour explicite (via `return`), exactement comme pour les routines. L'expression fournissant la valeur de retour est évaluée en contexte vide, scalaire ou de liste, en fonction du contexte

du eval elle-même. Voir `wantarray` pour de plus amples informations sur la façon dont le contexte d'évaluation peut être déterminé.

En cas d'erreur de syntaxe ou d'exécution ou si une instruction `die()` est exécutée, une valeur indéfinie (`undef`) est retournée par `eval()` et le message d'erreur est assigné à `$@`. S'il n'y a pas d'erreur, vous avez la garantie que la valeur de `$@` sera une chaîne de caractères vide. Prenez garde au fait qu'utiliser `eval()` ne dispense Perl ni d'afficher des messages d'alerte (`warnings`) sur `STDERR` ni d'assigner ses messages d'alerte dans `$@`. Pour ce faire, il vous faut utiliser les capacités de `$_SIG{__WARN__}` ou désactiver les messages d'avertissement dans le BLOC ou l'EXPR en utilisant `no warnings 'all'`. Voir `warn`, `perlvar`, `warnings` et `perlexwarn`.

Étant donné que `eval()` intercepte les erreurs non fatales, c'est très pratique pour déterminer si une fonctionnalité (telle que `socket()` ou `symlink()`) est supportée. C'est aussi le mécanisme d'interception d'exception de Perl lorsque l'opérateur `die` est utilisé pour les soulever.

Si le code à exécuter ne varie pas, vous devriez utiliser la seconde forme avec un BLOC pour intercepter les erreurs d'exécution sans supporter l'inconvénient de le recompiler à chaque fois. L'erreur, le cas échéant, est toujours retournée dans `$@`. Exemples :

```
rend une division par zéro non fatale
eval { $answer = $a / $b; }; warn $@ if $@;

même chose, mais moins efficace
eval '$answer = $a / $b'; warn $@ if $@;

une erreur de compilation
eval { $answer = }; # MAUVAIS

une erreur d'exécution
eval '$answer ='; # sets $@
```

En utilisant la forme `eval{}` pour une interception d'exception dans une bibliothèque, vous pourriez souhaiter ne pas exécuter une éventuelle sous-routine attachée à `__DIE__` par du code de l'utilisateur. Dans ce cas, vous pouvez utiliser la construction `local $_SIG{__DIE__}` comme dans l'exemple ci-dessous :

```
une interception d'exception de division par zéro très privée
eval { local $_SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

Ceci est spécialement significatif, étant donné que les sous-routines attachées à `__DIE__` peuvent appeler `die()` à nouveau, ce qui a pour effet de changer leurs messages d'erreur :

```
les sous-routines attachées à __DIE__
peuvent modifier les messages d'erreur
{
 local $_SIG{'__DIE__'} =
 sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
 eval { die "foo lives here" };
 print $@ if $@; # affiche "bar lives here"
}
```

Étant donné que cela favorise une action à distance, ce comportement contre-intuitif pourrait changer dans une version future.

Avec `eval()`, vous devriez faire particulièrement attention à ce qui est examiné, et quand :

```
eval $x; # CAS 1
eval "$x"; # CAS 2

eval '$x'; # CAS 3
eval { $x }; # CAS 4

eval "\$$x++"; # CAS 5
$$x++; # CAS 6
```

Les cas 1 et 2 ci-dessus se comportent de la même façon : ils exécutent le code inclus dans la variable `$x`. (Bien que le cas 2 ait des guillemets qui font se demander au lecteur ce qui se passe – réponse : rien.) Les cas 3 et 4 se comportent aussi de la même façon : ils exécutent le code `'$x'` qui ne fait rien que retourner la valeur de `$x`. (Le cas 4 est préférable pour des raisons purement visuelles mais aussi parce qu'il a l'avantage d'être compilé lors de la compilation plutôt que lors de l'exécution.) Le cas 5 est un endroit où vous *DEVRIEZ* normalement souhaiter

utiliser des guillemets, sauf que dans ce cas particulier, vous pouvez juste utiliser les références symboliques à la place, comme dans le cas 6.

Un `eval BLOC` n'est pas considéré comme une boucle. Par conséquent, on ne peut pas utiliser les instructions de contrôle de boucles `next`, `last`, ou `redo` pour quitter ou réexécuter le bloc.

Notez qu'il existe un cas vraiment très spécial : un `eval` " exécuté dans le package `DB` n'est pas dans la portée lexical de ce qui l'entoure mais dans celle du code le plus proche l'ayant appelé mais ne faisant pas partie de `DB`. Ceci n'a strictement aucun intérêt sauf si vous écrivez un débogueur Perl.

### exec LISTE

#### exec PROGRAMME LISTE

La fonction `exec()` exécute une commande système *et ne retourne jamais* – utilisez `system()` à la place de `exec()` si vous souhaitez qu'elle retourne. Elle échoue et retourne FAUX si et seulement si la commande n'existe pas *et* est exécutée directement plutôt que par votre interpréteur de commandes (shell) (cf. ci-dessous).

Comme c'est une erreur courante d'utiliser `exec()` au lieu de `system()`, Perl vous alerte si l'expression suivante n'est pas `die()`, `warn()`, ou `exit()` (si `-w` est utilisé – ce que vous faites toujours, évidemment.) Si vous voulez vraiment faire suivre le `exec()` d'une autre expression, vous pouvez utiliser l'une de ces méthodes pour éviter l'avertissement :

```
exec ('foo') or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

S'il y a plus d'un argument dans la LISTE, ou si la LISTE est un tableau de plus d'une valeur, appelle `execvp(3)` avec les arguments de la LISTE. S'il n'y a qu'un seul argument scalaire ou un tableau à un seul élément et que cet argument contient des méta-caractères du shell, l'argument entier est passé à l'interpréteur de commandes shell pour son expansion et son exécution (il s'agit de `/bin/sh -c` sur les plates-formes Unix, mais cela varie en fonction des plates-formes.) S'il n'y a aucun méta-caractères du shell dans l'argument, il est découpé en mots et passé directement à `execvp()` qui est plus efficace. Exemples :

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

Si vous ne souhaitez pas vraiment exécuter le premier argument mais plutôt l'utiliser pour dissimuler le nom du programme réellement utilisé, vous pouvez spécifier le vrai programme à exécuter comme un "objet indirect" (sans virgule) au début de la LISTE. (Ceci force toujours l'interprétation de la LISTE comme une liste multivaluée, même s'il n'y a qu'un seul scalaire dedans.) Exemple :

```
$shell = '/bin/csh';
exec $shell '-sh'; # prétend qu'il s'agit d'un login shell
```

ou, plus directement,

```
exec {'/bin/csh'} '-sh'; # prétend qu'il s'agit d'un login shell
```

Quand les arguments sont exécutés par le shell système, les résultats seront dépendants de ses caprices et de ses capacités. Voir 'CHAINE' in *perlop* pour plus de détails.

Utiliser un objet indirect avec `exec()` ou `system()` est aussi plus sûr. Cet usage force l'interprétation des arguments comme une liste multivaluée, même si elle ne contient qu'un seul élément. De cette façon, vous ne risquez rien des *jokers* du shell ou de la séparation des mots par des espaces.

```
@args = ("echo surprise");
system @args; # sujet à des échappement du shell
 # si @args == 1
system { $args[0] } @args; # sûr même avec une liste à un seul élément
```

La première version, celle sans l'objet indirect, exécute le programme *echo*, en lui passant "surprise" comme argument. La seconde version ne le fait pas – elle essaie d'exécuter un programme littéralement appelé "echo surprise", ne le trouve pas et assigne à  `$?`  une valeur non nulle indiquant une erreur.

Depuis la version v5.6.0, Perl essaie avant d'effectuer le `exec()` de vider tous les tampons des fichiers ouverts en écriture mais ce n'est pas le cas sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appeler la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts afin d'éviter toute perte de données.

Remarquez que `exec()` n'exécutera ni vos blocs `END` ni les méthodes `DESTROY` de vos objets.

### exists EXPR

Si `EXPR` spécifie un élément d'une table de hachage ou d'un tableau, retourne VRAI si cet élément existe, même si la valeur correspondante est indéfinie. L'élément n'est pas créé automatiquement s'il n'existe pas.

```

print "Exists\n" if exists $array{$key};
print "Defined\n" if defined $array{$key};
print "True\n" if $array{$key};

print "Exists\n" if exists $hash{$key};
print "Defined\n" if defined $hash{$key};
print "True\n" if $hash{$key};

print "Exists\n" if exists $array[$index];
print "Defined\n" if defined $array[$index];
print "True\n" if $array[$index];

```

Un élément d'une table de hachage ou d'un tableau ne peut être VRAI que s'il est défini, et défini que s'il existe, mais la réciproque n'est pas nécessairement vraie.

Si `EXPR` spécifie le nom d'une subroutine, retourne VRAI si la subroutine spécifiée a déjà été déclarée, même si elle est toujours indéfinie. Utiliser un nom de subroutine pour tester si elle existe (`exists`) ou si elle est définie (`defined`) ne compte pas comme une déclaration.

```

print "Exists\n" if exists &subroutine;
print "Defined\n" if defined &subroutine;

```

Notez que l'expression `EXPR` peut être arbitrairement compliquée tant qu'au final, elle désigne une subroutine ou un élément d'une table de hachage ou d'un tableau :

```

if (exists $ref->{A}->{B}->{$key}) { }
if (exists $hash{A}{B}{$key}) { }

if (exists $ref->{A}->{B}->[$ix]) { }
if (exists $hash{A}{B}[$ix]) { }

if (exists &{$ref->{A}{B}{$key}}) { }

```

Bien que l'élément le plus profond ne soit pas soudainement créé juste parce son existence a été testée, les éléments intermédiaires, eux, le seront. Par conséquent, `$ref->{"A"}` et `$ref->{"A"}->{"B"}` seront créés à cause du test d'existence de l'élément lié à la clé `$key` ci-dessus. Cela arrivera à chaque fois que l'opérateur flèche est utilisé, y compris dans le cas suivant :

```

undef $ref;
if (exists $ref->{"Some key"}) { }
print $ref; # affiche HASH(0x80d3d5c)

```

Cette génération spontanée un peu surprenante qui, au premier coup d'oeil (ni même au second d'ailleurs), n'est pas dans le contexte d'une lvalue pourrait être supprimée dans une version future.

Voir Pseudo-tables de hachage : utiliser un tableau comme table de hachage in *perlref* pour des informations spécifiques concernant l'utilisation de `exists()` sur les pseudo-hachages.

L'utilisation d'un appel à une subroutine à la place du nom de cette subroutine comme argument de `exists()` est une erreur.

```

exists ⊂ # OK
exists &sub(); # Erreur

```

## exit EXPR

### exit

Évalue `EXPR` puis quitte immédiatement avec cette valeur. Exemple :

```

$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;

```

Voir aussi `die()`. Si l'expression `EXPR` est omise, quitte avec le statut `0`. Les seules valeurs universellement reconnues pour `EXPR` sont `0` en cas de réussite et `1` en cas d'erreur ; toutes les autres valeurs sont sujettes à des interprétations imprévisibles, en fonction de l'environnement dans lequel le programme Perl est exécuté. Par exemple, terminer un filtre de messages entrants de *sendmail* avec comme valeur `69` (`EX_UNAVAILABLE`) provoquera la non livraison du message, mais ce n'est pas vrai partout.

Vous ne devriez pas utiliser `exit()` pour interrompre une routine s'il existe une chance pour que quelqu'un souhaite intercepter une erreur qui arrive. Utilisez `die()` à la place, qui peut être intercepté par un `eval()`.

La fonction `exit()` ne termine pas toujours le process immédiatement. Elle appelle d'abord toutes les routines `END` définies, mais ces routines `END` ne peuvent pas annuler la terminaison. De la même façon, tout destructeur d'objet qui doit être appelé le sera avant la sortie. Si cela pose problème, vous pouvez appelé `POSIX::_exit($status)` pour éviter le traitement des destructeurs et des subroutine `END`. Voir *perlmod* pour les détails.

**exp** **EXPR****exp**

Retourne  $e$  (la base des logarithmes naturels ou népérien) élevé à la puissance **EXPR**. Si **EXPR** est omis, retourne `exp($_)`.

**fcntl** **DESCRIPTEUR, FONCTION, SCALAIRE**

Implémente la fonction `fcntl(2)`. Vous devrez probablement d'abord écrire :

```
use Fcntl;
```

pour récupérer les définitions correctes des constantes. Le traitement de l'argument et la valeur de retour se fait exactement de la même manière que `ioctl()` plus bas. Par exemple :

```
use Fcntl;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
 or die "can't fcntl F_GETFL: $!";
```

Vous n'avez pas à vérifier le résultat de `fcntl` via `defined`. Comme `ioctl`, la valeur de retour `0` de l'appel système est transformée en un `"0 but true"` en Perl. Cette chaîne est vraie dans un contexte booléen et vaut `0` dans un contexte numérique. Elle est aussi exempte des alertes habituelles de `-w` sur les conversions numériques impropres. Notez que `fcntl()` produira une erreur fatale si elle est utilisée sur une machine n'implémentant pas `fcntl(2)`. Voir le module `Fcntl` ou `fcntl(2)` pour connaître les fonctions disponibles sur votre système.

Voici un exemple rendant non-bloquant le descripteur nommé `REMOTE`. Par contre, il vous restera à fixer `$!`.

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
 or die "Can't get flags for the socket: $!\n";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
 or die "Can't set flags for the socket: $!\n";
```

**fileno** **DESCRIPTEUR**

Retourne le numéro système associé à un descripteur de fichier. Ceci est utile pour construire des vecteurs pour `select()` et pour les opérations POSIX de manipulation bas niveau de terminaux `tty`. Si **DESCRIPTEUR** est une expression, la valeur est prise comme un descripteur indirect, généralement son nom.

Vous pouvez utiliser ceci pour déterminer si deux descripteurs se réfèrent au même numéro sous-jacent :

```
if (fileno(THIS) == fileno(THAT)) {
 print "THIS and THAT are dups\n";
}
```

(Les descripteurs connectés à des objets en mémoire via les nouvelles fonctionnalités de `open` peuvent retourner une valeur indéfinie bien qu'ils soient ouverts.)

**flock** **DESCRIPTEUR, OPERATION**

Appelle `flock(2)`, ou son émulation, sur le **DESCRIPTEUR**. Retourne **VRAI** en cas de succès, **FAUX** en cas d'échec. Produit une erreur fatale lorsqu'il est utilisé sur une machine n'implémentant pas `flock(2)`, le verrouillage `fcntl(2)`, ou `lockf(3)`. `flock()` est une interface Perl portable de verrouillage de fichiers, bien qu'il ne verrouille que des fichiers en entier, et non pas des enregistrements.

La sémantique non évidente mais néanmoins traditionnelle de `flock` consiste à attendre indéfiniment jusqu'à la libération du verrou. Ce verrou est **purement consultatif**. De tels verrous sont plus faciles d'utilisation mais offrent moins de garantie. Cela signifie que des programmes n'utilisant pas `flock` peuvent modifier des fichiers verrouillés via `flock`. Voir *perlport* ou les pages de manuel de votre système pour la documentation plus détaillée. Il vaut mieux faire avec ce comportement traditionnel si vous visez la portabilité. (Sinon, libre à vous d'utiliser les fonctionnalités ("features") de votre système d'exploitation. Les contraintes de portabilité ne doivent pas vous empêcher de faire ce que voulez.)

**OPERATION** peut être `LOCK_SH`, `LOCK_EX` ou `LOCK_UN`, éventuellement combinée avec `LOCK_NB`. Ces constantes ont traditionnellement pour valeurs 1, 2, 8 et 4, mais vous pouvez utiliser les noms symboliques s'ils sont importés du module `Fcntl`, soit individuellement, soit en tant que groupe en utilisant la balise `'flock'`. `LOCK_SH` demande un verrou partagé, `LOCK_EX` demande un verrou exclusif et `LOCK_UN` libère un verrou précédemment demandé. Si `LOCK_NB` est ajouté à `LOCK_SH` ou `LOCK_EX` (via un 'ou' bit à bit) alors `flock()` retournera immédiatement plutôt que d'attendre la libération du verrou (vérifiez le code de retour pour savoir si vous l'avez obtenu).

Pour éviter une mauvaise synchronisation, Perl vide le tampon du **DESCRIPTEUR** avant de le (dé)verrouiller.

Notez que l'émulation construite avec `lockf(3)` ne fournit pas de verrous partagés et qu'il exige que `DESCRIPTEUR` soit ouvert en écriture. Ce sont les sémantiques qu'implémente `lockf(3)`. La plupart des systèmes (si ce n'est tous) implémentent toutefois `lockf(3)` en termes de verrous `fcntl(2)`, la différence de sémantiques ne devrait donc pas gêner trop de monde.

Notez que l'émulation de `flock(3)` via `fcntl(2)` implique que `DESCRIPTEUR` soit ouvert en lecture pour l'utilisation de `LOCK_SH` et en écriture pour l'utilisation de `LOCK_EX`.

Notez aussi que certaines versions de `flock()` ne peuvent pas verrouiller des choses à travers le réseau, vous devriez utiliser des appels à `fcntl()` plus spécifiques au système dans ce but. Si vous le souhaitez, vous pouvez contraindre Perl à ignorer la fonction `flock(2)` de votre système et lui fournir ainsi sa propre émulation basée sur `fcntl(2)`, en passant le flag `-Ud_flock` au programme *Configure* lors de la configuration de perl.

Voici un empilage de mails pour les systèmes BSD.

```
use Fcntl ':flock'; # import des constantes LOCK_*

sub lock {
 flock(MBOX, LOCK_EX);
 # et, si quelqu'un ajoute
 # pendant notre attente...
 seek(MBOX, 0, 2);
}

sub unlock {
 flock(MBOX, LOCK_UN);
}

open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
 or die "Can't open mailbox: $!";

lock();
print MBOX $msg, "\n\n";
unlock();
```

Sur les systèmes qui possèdent un vrai `flock()`, les verrous sont hérités à travers les appels à `fork()` alors que ceux qui s'appuient sur la fonction `fcntl()` plus capricieuse perdent les verrous, rendant ainsi l'écriture de serveur plus difficile.

Voir aussi `DB_File` pour d'autres exemples avec `flock()`.

## fork

Effectue un appel système `fork(2)` pour créer un nouveau processus exécutant le même programme au même point. Retourne l'identifiant (pid) de l'enfant au processus père, 0 au processus fils ou `undef` en cas d'échec. Les descripteurs de fichiers (et parfois les verrous posés sur ces descripteurs) sont partagés sinon tout le reste est copié. Sur la plupart des systèmes qui supportent l'appel système `fork()`, une grande attention a été portée pour qu'il soit extrêmement efficace (par exemple en utilisant la méthode de copie-à-l-écriture sur les pages mémoires contenant des données). C'est donc la paradigme dominant pour la gestion du multi-tâches durant les dernières décennies.

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant d'effectuer le `fork()` mais cela n'est pas supporté sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appeler la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts afin d'éviter toute duplication de données.

Si vous dupliquez avec `fork()` sans attendre votre fils, vous allez accumuler des zombies. Sur certains systèmes, vous pouvez éviter cela en positionnant `$SIG{CHLD}` à "IGNORE". Voir aussi *perlipc* pour des exemples d'utilisation de `fork()` et de suppression d'enfants moribonds.

Notez que si votre fils dupliqué hérite de descripteurs de fichier systèmes, tels que `STDIN` et `STDOUT`, qui sont en fait connectés par un tube ou une socket, même si vous sortez du programme, alors le serveur distant (disons tels que `httpd` ou `rsh`) ne saura pas que vous avez terminé. Vous devriez les réouvrir vers `/dev/null` en cas de problème.

## format

Déclare un format visuel utilisable par la fonction `write()`. Par exemple :

```
format Something =
 Test: @<<<<<<<< @| || || @>>>>>>
 $str, $%, '$' . int($num)
.
```

```

$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
write;

```

Voir *perlfom* pour de nombreux détails et exemples.

### formline IMAGE,LISTE

C'est une fonction interne utilisée par les formats (*format*), bien que vous puissiez aussi l'appeler. Elle formate (voir *perlfom*) une liste de valeurs selon le contenu de *IMAGE*, plaçant la sortie dans l'accumulateur du format de sortie  $\$^A$  (ou *\$ACCUMULATOR* en anglais). Finalement, lorsqu'un *write()* est effectué, le contenu de  $\$^A$  est écrit dans un descripteur de fichier. Vous pouvez aussi lire  $\$^A$  et réassigner "" à  $\$^A$ . Notez qu'un format typique appelle *formline* une fois par ligne du formulaire mais la fonction *formline* elle-même ne se préoccupe pas du nombre de passage à la ligne inclus dans l'*IMAGE*. Ceci signifie que *~* et *~~* traiteront l'*IMAGE* complète comme une seule ligne. Vous pouvez donc utiliser de multiples *formline* pour implémenter un seul format d'enregistrement, tout comme le compilateur de format.

Faites attention si vous utilisez des guillemets autour de l'image, car un caractère "@" pourrait être pris pour le début d'un nom de tableau. *formline()* retourne toujours VRAI. Cf. *perlfom* pour d'autres exemples.

### getc DESCRIPTEUR

#### getc

Retourne le prochain caractère du fichier d'entrée attaché au DESCRIPTEUR ou la valeur indéfinie (*undef*) à la fin du fichier ou en cas d'erreur (en fixant *#!* dans ce dernier cas). Si le DESCRIPTEUR est omis, utilise *STDIN*. Ce n'est pas particulièrement efficace. De plus, ce n'est pas utilisable tel quel pour obtenir des caractères un à un sans que l'utilisateur presse *ENTER*. Pour ça, essayez quelque chose comme :

```

if ($BSD_STYLE) {
 system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
 system "stty", '-icanon', 'eol', "\001";
}

$key = getc(STDIN);

if ($BSD_STYLE) {
 system "stty -cbreak </dev/tty >/dev/tty 2>&1";
}
else {
 system "stty", 'icanon', 'eol', '^@'; # ASCII null
}
print "\n";

```

Déterminer la valeur de *\$BSD\_STYLE* est laissé en exercice au lecteur.

La fonction *POSIX::getattr()* peut faire ceci de façon plus portable sur des systèmes compatibles *POSIX*. Voir aussi le module *Term::ReadKey* de votre site CPAN le plus proche. Les détails sur CPAN peuvent être trouvés dans CPAN in *perlmodlib*.

#### getlogin

Implémente la fonction de la bibliothèque C portant le même nom et qui, sur la plupart des systèmes, retourne le login courant à partir de */etc/utmp*, si il existe. Si l'appel retourne null, utilisez *getpuid()*.

```
$login = getlogin || getpuid($<) || "Kilroy";
```

N'utilisez pas *getlogin()* pour de l'authentification : ce n'est pas aussi sûr que *getpuid()*.

#### getpeername SOCKET

Renvoie l'adresse *sockaddr* compactée (voir *pack()*) de l'autre extrémité de la connexion *SOCKET*.

```

use Socket;
$hersockaddr = getpeername(SOCK);
($port, $iaddr) = unpack_sockaddr_in($hersockaddr);
$herhostname = gethostbyaddr($iaddr, AF_INET);
$herstraddr = inet_ntoa($iaddr);

```

**getpgrp PID**

Renvoie le groupe courant du processus dont on fournit le PID. Utilisez le PID 0 pour obtenir le groupe courant du processus courant. Cela engendra une exception si on l'utilise sur une machine qui n'implémente pas `getpgrp(2)`. Si PID est omis, renvoie le groupe du processus courant. Remarquez que la version POSIX de `getpgrp()` ne prend pas d'argument PID donc seul `PID==0` est réellement portable.

**getppid**

Renvoie l'id du processus parent.

Note pour les utilisateurs Linux : sur Linux, les fonctions C `getppid()` et `getpid()` retournent des valeurs différentes selon les fils d'exécution (les différents threads). Pour être portable, ce comportement n'est pas celui de la fonction perl `getppid()` qui retourne une valeur identique pour tous les fils (threads) d'un même processus. Pour appeler la vraie fonction `getppid()`, vous pouvez utiliser le module CPAN `Linux::Pid`.

**getpriority WHICH,WHO**

Renvoie la priorité courant d'un processus, d'un groupe ou d'un utilisateur. (Voir `getpriority(2)`.) Cela engendra une exception si on l'utilise sur une machine qui n'implémente pas `getpriority(2)`.

**getpwnam NAME****getgrnam NAME****gethostbyname NAME****getnetbyname NAME****getprotobyname NAME****getpwuid UID****getgrgid GID****getservbyname NAME,PROTO****gethostbyaddr ADDR,ADDRTYPE****getnetbyaddr ADDR,ADDRTYPE****getprotobynumber NUMBER****getservbyport PORT,PROTO****getpwent****getgrent****gethostent****getnetent****getprotoent****getservent****setpwent****setgrent****sethostent STAYOPEN****setnetent STAYOPEN****setprotoent STAYOPEN****setservent STAYOPEN****endpwent****endgrent****endhostent****endnetent****endprotoent****endservent**

Ces routines réalisent exactement les mêmes fonctions que leurs homologues de la bibliothèque système. Dans un contexte de liste, les valeurs retournées par les différentes routines sont les suivantes :

```
($name, $passwd, $uid, $gid,
 $quota, $comment, $gcos, $dir, $shell, $expire) = getpw*
($name, $passwd, $gid, $members) = getgr*
($name, $aliases, $addrtype, $length, @addrs) = gethost*
($name, $aliases, $addrtype, $net) = getnet*
($name, $aliases, $proto) = getproto*
($name, $aliases, $port, $proto) = getserv*
```



(Si une entrée n'existe pas, vous récupérerez une liste vide.)

La signification exacte du champ `$gc` varie mais contient habituellement le nom réel de l'utilisateur (au contraire du nom de login) et d'autres informations pertinentes pour cet utilisateur. Par contre, sachez que sur de nombreux systèmes, les utilisateurs peuvent changer eux-mêmes ces informations. Ce n'est donc pas une information de confiance. Par conséquent `$gc` est souillée (voir *perlsec*). Les champs `$passwd`, `$shell` ainsi que l'interpréteur de commandes (login shell) et le mot de passe crypté sont aussi souillés pour les mêmes raisons.

Dans un contexte scalaire, vous obtenez le nom sauf lorsque la fonction fait une recherche par nom auquel cas vous récupérerez autre chose. (Si une entrée n'existe pas vous récupérerez la valeur `undef`.) Par exemple :

```
$uid = getpwnam($name);
$name = getpwuid($num);
$name = getpwent();
$gid = getgrnam($name);
$name = getgrgid($num);
$name = getgrent();
#etc.
```

Dans *getpw\**(*)*, les champs `$quota`, `$comment` et `$expire` sont des cas spéciaux dans le sens où ils ne sont pas supportés sur de nombreux systèmes. Si `$quota` n'est pas supporté, c'est un scalaire vide. Si il est supporté, c'est habituellement le quota disque. Si le champ `$comment` n'est pas supporté, c'est un scalaire vide. Si il est supporté, c'est habituellement le commentaire « administratif » associé à l'utilisateur. Sur certains systèmes, le champ `$quota` peut être `$change` ou `$age`, des champs qui sont en rapport avec le vieillissement du mot de passe. Sur certains systèmes, le champ `$comment` peut être `$class`. Le champ `$expire`, s'il est présent, exprime la période d'expiration du compte ou du mot de passe. Pour connaître la disponibilité et le sens exact de tous ces champs sur votre système, consultez votre documentation de *getpwnam*(3) et le fichier *pwd.h*. À partir de Perl, vous pouvez trouver le sens de vos champs `$quota` et `$comment` et savoir si vous avez le champ `$expire` en utilisant le module *Config* pour lire les valeurs de `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment` et `d_pwexpire`. Les fichiers de mots de passe cachés (shadow password) ne sont supportés que si l'implémentation de votre système est faite telle que les appels aux fonctions normales de la bibliothèque C accèdent à ces fichiers lorsque vos privilèges vous y autorisent ou lorsque les fonctions *shadow*(3) existent comme sous System V (ce qui est le cas de Solaris et de Linux). Toute autre implémentation de ces fonctionnalités via des appels à une bibliothèque spécifique n'est pas supportée.

La valeur `$members` renvoyée par les fonctions *getgr\**(*)* est la liste des noms de login des membres du groupe séparés par des espaces.

Pour les fonctions *gethost\**(*)*, si la variable `h_errno` est supportée en C, sa valeur sera retournée via  `$?`  si l'appel à la fonction échoue. La valeur de `@addr` qui est retournée en cas de succès est une liste d'adresses à plat telle que retournée par l'appel système correspondant. Dans le domaine Internet, chaque adresse fait quatre octets de long et vous pouvez la décompacter en disant quelque chose comme :

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

Si vous êtes fatigué de devoir vous souvenir que tel élément de la liste retournée correspond à telle valeur, des interfaces par nom sont fournies par les modules `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime` et `User::grent`. Ils remplacent les appels internes normaux par des versions qui renvoient des objets ayant le nom approprié pour chaque champ. Par exemple :

```
use File::stat;
use User::pwent;
$his = (stat($filename)->uid == pwent($whoever)->uid);
```

Bien que les deux appels se ressemblent (appel à la méthode `'uid'`), ce n'est pas la même chose car l'objet `File::stat` est différent de l'objet `User::pwent`.

### getsockname SOCKET

Renvoie l'adresse `sockaddr` compactée de cette extrémité de la connexion *SOCKET* si vous ne connaissez pas cette adresse car vous avez différentes adresses IP utilisables pour établir cette connexion.

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
 scalar gethostbyaddr($myaddr, AF_INET),
 inet_ntoa($myaddr);
```

**getsockopt SOCKET,LEVEL,OPTNAME**

Récupère l'option nommée OPTNAME associé au socket SOCKET au niveau LEVEL. Les options peuvent être utilisées à différents niveaux selon les protocoles liés au socket mais elles existent toujours au niveau du socket lui-même, le niveau SOL\_SOCKET (défini par le module Socket). Pour accéder à une option d'un autre niveau, vous devez fournir le numéro du protocole qui contrôle cette option. Par exemple, pour retrouver une option liée au protocole TCP, LEVEL doit être le numéro du protocole TCP que vous pouvez retrouver via getprotobyname.

L'appel retourne un chaîne compactée (par pack) représentant l'option de socket demandée ou undef en cas d'erreur (la raison de l'erreur est dans \$!). Ce que contient la chaîne dépend de LEVEL et de OPTNAME. Consultez la documentation de votre système pour les détails. Dans de nombreux cas, la valeur de l'option est un entier et vous pourrez décoder la chaîne compactée obtenue en utilisant unpack avec le format i (ou I).

Voici un exemple permettant de tester si l'algorithme Nagle est actif (on ou off) sur un socket :

```
use Socket qw(:all);

defined(my $tcp = getprotobyname("tcp"))
 or die "Could not determine the protocol number for tcp";
my $tcp = IPPROTO_TCP; # Alternative
my $packed = getsockopt($socket, $tcp, TCP_NODELAY)
 or die "Could not query TCP_NODELAY socket option: $!";
my $nodelay = unpack("I", $packed);
print "Nagle's algorithm is turned ", $nodelay ? "off\n" : "on\n";
```

**glob EXPR****glob**

Dans un contexte de liste, retourne la liste (éventuellement vide) des fichiers correspondants à l'expansion de la valeur de EXPR telle que le shell standard Unix */bin/csh* la ferait. Dans un contexte scalaire, glob produit les éléments de cette liste un par un jusqu'à l'épuiser puis il retourne undef. C'est la fonction interne qui implémente l'opérateur `<*.c>` mais vous pouvez l'utiliser directement. Si EXPR est omis, \$\_ est utilisé à la place. L'opérateur `<*.c>` est présenté plus en détail dans Les opérateurs d'E/S in *perlop*.

Depuis la version v5.6.0, cet opérateur est implémenté via l'extension standard `File::Glob`. Voir `File::Glob` pour plus de détails.

**gmtime EXPR****gmtime**

Convertit une date telle que celle retournée par la fonction time en un tableau de 9 éléments en prenant comme référence le fuseau horaire standard du méridien de Greenwich. On l'utilise typiquement de la manière suivante :

```
0 1 2 3 4 5 6 7 8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$ajour,$isdst) =
 gmtime(time);
```

Tous les éléments du tableau sont numériques et restent tels qu'ils apparaissent dans la structure 'struct tm'. \$sec, \$min et \$heure sont les secondes, les minutes et l'heure de l'instant spécifié. \$mjour est le quantième du mois et \$mois est le mois lui-même, dans l'intervalle 0..11 avec 0 pour janvier et 11 pour décembre. \$annee est le nombre d'années depuis 1900 et donc \$annee vaut 123 en l'an 2023. \$sjour est le jour de la semaine avec 0 pour le dimanche et 3 pour le mercredi. \$ajour est le jour de l'année dans l'intervalle 1..365 (ou 1..366 pour les années bissextiles.) \$isdst vaut toujours 0.

Remarquez bien que \$annee n'est pas que les deux derniers chiffres de l'année. Si vous supposez cela, vous créez des programmes non compatible an 2000 – et vous ne voulez pas faire cela, n'est-ce pas ?

La bonne méthode pour obtenir une année complète sur 4 chiffres est tout simplement :

```
$annee += 1900;
```

Et pour obtenir les deux derniers chiffres de l'année (e.g. '01' en 2001):

```
$annee = sprintf("%02d", $annee % 100);
```

Si EXPR est omis, calcule gmtime(time).

Dans un contexte scalaire, retourne la valeur de ctime(3) :

```
$now_string = gmtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

Si vous souhaitez une date exprimée selon le fuseau horaire local, utilisez `localtime`. Voir aussi la fonction `timegm()` fournit par le module `Time::Local` et les fonctions `strftime(3)` et `mktime(3)` disponibles via le module `POSIX`.

La valeur scalaire n'est **pas** dépendante du locale, voir *perllocale*, mais est construite en interne par Perl. Pour obtenir quelque chose de similaire mais dont les chaînes de caractères des dates dépendent du locale, voir les exemples dans *perllocale*.

Voir `gmtime` in *perlport* pour les aspects liés à la portabilité.

#### **goto LABEL**

#### **goto EXPR**

#### **goto &NAME**

La forme `goto-LABEL` trouve la ligne étiquetée par `LABEL` et continue l'exécution à cette ligne. On ne peut pas l'utiliser pour sauter à l'intérieur d'une construction qui nécessite une initialisation telle qu'une subroutine ou une boucle `foreach`. On ne peut pas non plus l'utiliser pour aller dans une construction qui peut être optimisée ou pour sortir d'un bloc ou d'une subroutine donné à `sort()`. On peut par contre l'utiliser pour aller n'importe où dans la portée actuelle, même pour sortir des subroutines, bien qu'il soit meilleur d'utiliser pour cela d'autres constructions telles que `last` ou `die()`. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl bien sûr – en C, c'est une autre histoire). (La différence avec le C provient du fait que le C n'offre pas le nommage des boucles couplée aux instructions de contrôle de boucles. Cette possibilité de Perl remplace la quasi totalité des usages structurés du `goto` dans d'autres langages.)

La forme `goto-EXPR` attend un nom d'étiquette dont la portée sera résolue dynamiquement. Cela autorise les `goto` calculés à la FORTRAN mais ce n'est pas vraiment recommandé si vous vous souciez de la maintenance.

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

La forme `goto-&NAME` est vraiment différente des autres formes de `goto`. En fait, ce n'est pas du tout un `goto` et il n'a donc pas les stigmates habituellement associés aux autres `goto`. Cette forme substitue l'appel de la subroutine en cours d'exécution (en oubliant toutes les modifications faites par des appels à `local()`) par un appel à la subroutine donnée en argument. C'est utilisé par les routines `AUTOLOAD` qui désirent charger une autre subroutine puis tout faire comme si c'était cette autre subroutine qui avait réellement été appelée (sauf que toutes les modifications faites à `@_` dans la subroutine courante sont propagées à l'autre subroutine). Après le `goto`, pas même `caller()` n'est capable de s'apercevoir que la subroutine initiale a été appelée au préalable.

`NAME` n'est pas nécessairement le nom d'une subroutine ; cela peut être une variable scalaire contenant une référence à du code ou un bloc dont l'évaluation produit une référence à du code.

#### **grep BLOC LISTE**

#### **grep EXPR,LISTE**

Cette fonction est similaire dans l'esprit mais pas identique à `grep(1)` et tous ses dérivés. En particulier, elle n'est pas limitée à l'utilisation d'expressions rationnelles.

Elle évalue le bloc `BLOC` ou l'expression `EXPR` pour chaque élément de `LISTE` (qui est localement lié à `$_`) et retourne la liste des valeurs constituée des éléments pour lesquels l'expression est évaluée à `true` (vrai). Dans un contexte scalaire, retourne le nombre de fois où l'expression est vraie (`true`).

```
@foo = grep(!/^#/ , @bar); # supprime les commentaires
```

ou de manière équivalente :

```
@foo = grep {!/^#/} @bar; # supprime les commentaires
```

Remarquez que, puisque `$_` est un référence dans la liste de valeurs, il peut être utilisé pour modifier les éléments du tableau. Bien que ce soit supporté et parfois pratique, cela peut aboutir à des résultats bizarres si `LISTE` n'est pas un tableau nommé. De manière similaire, `grep` renvoie des alias de la liste originale exactement comme le fait une variable de boucle `for`. Et donc, modifier un élément d'une liste retournée par `grep` (par exemple dans un `foreach`, un `map()` ou un autre `grep()`) modifie réellement l'élément de la liste originale.

Voir aussi `map` pour obtenir un tableau composé des résultats de `BLOC` ou `EXPR`.

#### **hex EXPR**

#### **hex**

Interprète `EXPR` comme une chaîne hexadécimale et retourne la valeur correspondante. (Pour convertir des chaînes qui commencent par `0`, `0x` ou `0b`, voir `oct`.) Si `EXPR` est omis, c'est `$_` qui est utilisé.

```
print hex '0xAf'; # affiche '175'
print hex 'aF'; # idem
```

Les chaînes hexadécimales ne peuvent représenter que des entiers. Les chaînes qui provoquent un dépassement de la capacité des entiers déclenchent un avertissement. Au contraire de `oct()`, les espaces initiaux ne sont pas ignorés. Pour afficher quelque chose en hexadécimale, jeter un oeil sur `printf`, `sprintf` ou `unpack`.

### import LISTE

Il n'existe pas de fonction interne `import()`. C'est juste une méthode ordinaire (une sous-routine) définie (ou héritée) par les modules qui veulent exporter des noms vers d'autres modules. La fonction `use()` appelle la méthode `import()` du package utilisé. Voir aussi `use()`, *perlmod* et *Exporter*.

### index CHAINE,SUBSTR,POSITION

#### index CHAINE,SUBSTR

La fonction `index()` recherche une chaîne dans une autre mais sans les fonctionnalités génériques de reconnaissance de motifs des expressions rationnelles. Retourne la position de la première occurrence du `SUBSTR` dans `CHAINE` à partir de `POSITION` inclus. Si `POSITION` est omis, le recherche commence au début de la chaîne. Si `POSITION` est avant le début ou après la fin de la chaîne, utilisera respectivement le début ou la fin de la chaîne. La valeur retournée est relative à `0` (ou à la valeur de référence que vous avez affectée à la variable `$[` – mais ce n'est pas à faire). Si la sous-chaîne `SUBSTR` n'est pas trouvée, `index` retournera la valeur de référence moins 1 (habituellement -1).

### int EXPR

#### int

Retourne la partie entière de `EXPR`. Si `EXPR` est omis, c'est `$_` qui est utilisé. Vous ne devriez pas utiliser cette fonction pour faire des arrondis parce qu'elle tronque la valeur vers `0` et parce que la représentation interne des nombres en virgule flottante produit parfois des résultats contre-intuitifs. Par exemple `int(-6.725/0.025)` produit -268 au lieu de -269; C'est dû aux erreurs de calcul qui donne un résultat comme -268.99999999999994315658. Habituellement `sprintf()` et `printf()` ou les fonctions POSIX::`floor` et POSIX::`ceil` vous seront plus utiles que `int()`.

### ioctl DESCRIPTEUR,FONCTION,SCALAIRE

Implémente la fonction `ioctl(2)`. Vous aurez probablement à dire :

```
require "sys/ioctl.ph";
probablement dans $Config{archlib}/sys/ioctl.ph
```

en premier lieu pour obtenir les définitions correctes des fonctions. Si `sys/ioctl.ph` n'existe pas ou ne donne pas les définitions correctes, vous devrez les fournir vous-même en vous basant sur les fichiers d'en-tête C tels que `<sys/ioctl.h>`. (Il existe un script Perl appelé **h2ph** qui vient avec le kit Perl et qui devrait vous aider à faire cela mais il n'est pas trivial.) `SCALAIRE` sera lu et/ou modifié selon la fonction `FONCTION` – un pointeur sur la valeur alphanumérique de `SCALAIRE` est passé comme troisième argument du vrai appel système `ioctl()`. (Si `SCALAIRE` n'a pas de valeur alphanumérique mais a une valeur numérique, c'est cette valeur qui sera passée plutôt que le pointeur sur la valeur alphanumérique. Pour garantir que c'est bien ce qui se passera, ajouter `0` au scalaire avant de l'utiliser.) Les fonction `pack()` et `unpack()` permettent de manipuler les différentes structures utilisées par `ioctl()`.

Les différentes valeurs retournées par `ioctl` (et `fcntl`) sont les suivantes :

si l'OS retourne :	alors Perl retournera :
-1	la valeur indéfinie ( <code>undef</code> )
0	la chaîne " <code>0 but true</code> "
autre nombre	ce nombre

Ainsi Perl retourne vrai en cas de succès et faux en cas d'échec et vous pouvez encore déterminer la véritable valeur retournée par le système d'exploitation :

```
$retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

La chaîne spéciale "`0 but true`" est traitée par `-w` comme une exception qui ne déclenche pas le message d'avertissement concernant les conversions numériques incorrectes.

### join EXPR,LISTE

Concatène les différentes chaînes de `LISTE` en une seule chaîne où les champs sont séparés par la valeur de `EXPR` et retourne cette nouvelle chaîne. Exemple :

```
$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

Au contraire de `split`, `join` ne prend pas un motif comme premier argument. À comparer à `split`.

**keys HASH**

Retourne une liste constituée de toutes les clés (en anglais : keys) de la table de hachage fournie. (Dans un contexte scalaire, retourne le nombre de clés.)

Les clés sont produites dans un ordre apparemment aléatoire. Cet ordre particulier pourrait changer dans une version future mais il est garanti que ce sera toujours le même que celui produit par les fonctions `values()` ou `each()` (en supposant que la table de hachage n'a pas été modifiée). Depuis Perl 5.8.1, cet ordre varie d'une exécution à l'autre pour des raisons de sécurité (voir *Attaques par complexité algorithmique* in *perlsec*).

Un effet de bord d'un appel à `keys()` est la réinitialisation de l'itérateur de HASH (voir `each`). En particulier, un appel à `keys()` dans un contexte vide a pour seul effet la réinitialisation de l'itérateur.

Voici encore une autre manière d'afficher votre environnement :

```
@keys = keys %ENV;
@values = values %ENV;
while (@keys) {
 print pop(@keys), '=', pop(@values), "\n";
}
```

et comment trier tout cela par ordre des clés :

```
foreach $key (sort(keys %ENV)) {
 print $key, '=', $ENV{$key}, "\n";
}
```

Pour trier une table de hachage par valeur, vous aurez à utiliser la fonction `sort()`. Voici le tri d'une table de hachage par ordre décroissant de ses valeurs :

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
 printf "%4d %s\n", $hash{$key}, $key;
}
```

En tant que `lvalue` (valeur modifiable), `keys()` vous permet de d'augmenter le nombre de réceptacles alloués pour la table de hachage concernée. Cela peut améliorer les performances lorsque vous savez à l'avance qu'une table va grossir. (C'est tout à fait similaire à l'augmentation de taille d'un tableau en affectant une grande valeur à  `$#tableau`.) Si vous dites :

```
keys %hash = 200;
```

alors `%hash` aura au moins 200 réceptacles alloués – 256 en fait, puisque la valeur est arrondie à la puissance de deux immédiatement supérieure. Ces réceptacles seront conservés même si vous faites `%hash = ()`. Utilisez `undef %hash` si vous voulez réellement libérer l'espace alloué. En revanche, vous ne pouvez pas utiliser cette méthode pour réduire le nombre de réceptacles alloués (n'ayez aucune inquiétude si vous le faites tout de même par inadvertance : cela n'a aucun effet).

Voir aussi `each`, `values` et `sort`.

**kill SIGNAL, LISTE**

Envoie un signal à une liste de processus. Retourne le nombre de processus qui ont été correctement « signalés » (qui n'est pas nécessairement le même que le nombre de processus à qui le signal a été envoyé).

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

Si `SIGNAL` vaut zéro, aucun signal n'est envoyé. C'est un moyen pratique de vérifier qu'un processus enfant existe encore et n'a pas changé son UID. Voir *perlport* pour vérifier la portabilité d'une telle construction.

Au contraire du shell, en Perl, si `SIGNAL` est négatif, il « kill » le groupe de processus plutôt que les processus. (Sur System V, un numéro de *PROCESSUS* négatif « kill » aussi les groupes de processus mais ce n'est pas portable.) Cela signifie que vous utiliserez habituellement une valeur positive comme signal. Vous pouvez aussi utiliser un nom de signal entre apostrophes.

Voir *Signaux* in *perlipc* pour tous les détails.

**last LABEL****last**

La commande `last` est comme l'instruction `break` en C (telle qu'elle est utilisée dans les boucles) ; cela permet de sortir immédiatement de la boucle en question. En l'absence de `LABEL`, la commande se réfère à la boucle englobante la plus profonde. Le bloc continue, s'il existe, n'est pas exécuté :

```

LINE: while (<STDIN>) {
 last LINE if /^$/; # exit when done with header
 #...
}

```

`last` ne peut pas être utilisé pour sortir d'un bloc qui doit retourner une valeur comme `eval {}`, `sub {}` ou `do {}` et ne devrait pas être utilisé pour sortir d'une opération `grep()` ou `map()`.

Notez qu'un bloc en lui-même est sémantiquement équivalent à une boucle qui ne s'exécuterait qu'une seule fois. Par conséquent, `last` peut être utilisé pour sortir prématurément d'un tel bloc.

Voir aussi `continue` pour une illustration du comment marche `last`, `next` et `redo`.

## lc EXPR

### lc

Retourne une version de `EXPR` entièrement en minuscules. C'est la fonction interne qui implémente le séquence d'échappement `\L` dans les chaînes entre guillemets. Respecte le locale courant `LC_CTYPE` si `use locale` est actif. Voir *perllocale* et *perlunicode* pour des plus amples informations concernant la gestion des "locale" et d'Unicode.

En l'absence de `EXPR`, s'applique à `$_`.

## lcfirst EXPR

### lcfirst

Retourne un version de `EXPR` avec le premier caractère en minuscule. C'est la fonction interne qui implémente le séquence d'échappement `\l` dans les chaînes entre guillemets. Respecte le locale courant `LC_CTYPE` si `use locale` est actif. Voir *perllocale* et *perlunicode* pour des plus amples informations concernant la gestion des "locale" et d'Unicode.

En l'absence de `EXPR`, s'applique à `$_`.

## length EXPR

### length

Retourne la longueur en *caractères* de la valeur de `EXPR`. En l'absence de `EXPR`, s'applique à `$_`. Notez que cette fonction ne s'applique ni à un tableau ni à une table de hachage pour savoir combien d'éléments ils contiennent. Pour cela, utilisez respectivement `scalar @tableau` et `scalar keys %hash`

Vous avez noté le mot *caractères* : si `EXPR` est en Unicode, vous récupérerez le nombre de caractères et non le nombre d'octets. Pour obtenir une longueur en octets, utilisez `do { use bytes; length(EXPR) }`. Voir *bytes*.

## link OLDFILE,NEWFILE

Créer un nouveau fichier `NEWFILE` lié à l'ancien fichier `OLDFILE`. Retourne `true` (vrai) en cas de succès ou `false` (faux) sinon.

## listen SOCKET,QUEUESIZE

Fait exactement la même chose que l'appel système du même nom. Retourne `true` (vrai) en cas de succès ou `false` (faux) sinon. Voir les exemples dans *Sockets : Communication Client/Serveur* in *perlipc*.

## local EXPR

Vous devriez certainement utiliser `my()` à la place car `local()` n'a pas la sémantique que la plupart des gens accorde à la notion « local ». Voir *Variables Privées via my()* in *perlsub* pour plus de détails.

`local()` modifie les variables listées pour qu'elles soient locales au bloc/fichier/eval englobant. Si plus d'une valeur est donnée, la liste doit être placée entre parenthèses. Voir *Valeurs Temporaires via local()* in *perlsub* pour plus de détails, en particulier tout ce qui touche aux tables de hachage et aux tableaux liés (par `tie()`).

## localtime EXPR

### localtime

Convertit une date telle que retournée par la fonction `time` en un tableau de 9 éléments avec la date liée au fuseau horaire local. On l'utilise typiquement de la manière suivante :

```

0 1 2 3 4 5 6 7 8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$jour,$isdst) =
 localtime(time);

```

Tous les éléments du tableau sont numériques et restent tels qu'ils apparaissent dans la structure 'struct tm'. `$sec`, `$min` et `$heure` sont les secondes, les minutes et l'heure de l'instant spécifié.

`$mjour` est le quantième du mois et `$mois` est le mois lui-même, dans l'intervalle 0..11 avec 0 pour janvier et 11 pour décembre. Cela rend plus facile l'accès aux noms des mois stockés dans une liste :

```
my @abbr = qw(Jan Fév Mar Avr Mai Jui Jui Aoû Sep Oct Nov Déc);
print "$mjour $abbr[$mois]";
$mois=9, $mjour=18 donne "10 Oct"
Notez, en français, le problème entre Jui(n) et Jui(11et)
```

\$annee est le nombre d'années depuis 1900 et donc \$annee vaut 123 en l'an 2023. La bonne méthode pour obtenir une année complète sur 4 chiffres est tout simplement :

```
$annee += 1900;
```

Et pour obtenir les deux derniers chiffres de l'année (ex. '01' en 2001):

```
$annee = sprintf("%02d", $annee % 100);
```

\$sjour est le jour de la semaine avec 0 pour le dimanche et 3 pour le mercredi. \$ajour est le jour de l'année dans l'intervalle 1..365 (ou 1..366 pour les années bissextiles.)

\$isdst est vrai si l'heure d'été est en cours à la date spécifiée et faux sinon.

En l'absence de `EXPR`, `localtime()` utilise la date courante (`localtime(time)`).

Dans un contexte scalaire, retourne la valeur de `ctime(3)` :

```
$now_string = localtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

La valeur scalaire n'est **pas** dépendante du locale mais construite en interne par Perl. Pour l'heure GMT et non celle du fuseau horaire local, utilisez la fonction interne `gmtime`. Voir aussi le module `Time::Local` (pour convertir des secondes, minutes, heures... en une valeur entière telle que retournée par `time()`) et les fonctions `strftime(3)` et `mktime(3)` du module `POSIX`.

Pour obtenir quelque chose de similaire mais dont les chaînes de caractères des dates dépendent du locale, paramétrez vos variables d'environnement liées au locale (voir *perllocale*) et essayez par exemple :

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
ou pour l'heure GMT tenant compte de votre locale
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Notez que `%a` et `%b`, les formes courtes du jour de la semaine et du mois de l'année, n'ont pas obligatoirement 3 caractères de long.

Voir `localtime` in *perlport* pour les aspects liés à la portabilité.

### lock TRUC

Cette fonction place un verrou coopératif sur une variable partagée ou sur l'objet référencé par `TRUC` pour la durée de la portée de ce lock.

`lock()` est un « mot clé faible » : cela signifie que si vous définissez une fonction par ce nom (avant tout appel à `lock()`), c'est cette fonction qui sera appelée. (En revanche, si vous dites `use threads` alors `lock()` est toujours un mot clé.) Voir *threads*.

### log EXPR

#### log

Retourne le logarithme népérien ou naturel (base  $e$ ) de `EXPR`. En l'absence de `EXPR`, retourne le log de `$_`. Pour obtenir le logarithme dans une autre base, utilisez la propriété suivante : le logarithme en base  $N$  d'un nombre est égal au logarithme naturel de ce nombre divisé par le logarithme naturel de  $N$ . Par exemple :

```
sub log10 {
 my $n = shift;
 return log($n)/log(10);
}
```

Voir `exp` pour l'opération inverse.

### lstat EXPR

#### lstat

Fait la même chose que la fonction `stat()` (y compris de modifier le descripteur spécial `_`) mais fournit les données du lien symbolique au lieu de celles du fichier pointé par le lien. Si les liens symboliques ne sont pas implémentés dans votre système, un appel normal à `stat()` est effectué. Pour plus de détails, voir la documentation de `stat`.

En l'absence de `EXPR`, utilise `$_`.

### m//

L'opérateur de correspondance (d'expressions rationnelles). Voir *perlop*.

**map BLOC LISTE****map EXPR,LISTE**

Évalue le bloc BLOC ou l'expression EXPR pour chaque élément de LISTE (en affectant localement chaque élément à `$_`) et retourne la liste de valeurs constituée de tous les résultats de ces évaluations. L'évaluation du bloc BLOC ou de l'expression EXPR a lieu dans un contexte de liste si bien que chaque élément de LISTE peut produire zéro, un ou plusieurs éléments comme valeur retournée.

```
@chars = map(chr, @nums);
```

transcrit une liste de nombres vers les caractères correspondants. Et :

```
%hash = map { getkey($_) => $_ } @array;
```

est juste une manière rigolote de dire :

```
%hash = ();
foreach $_ (@array) {
 $hash{getkey($_)} = $_;
}
```

Remarquez que du fait que `$_` est une référence vers la liste de valeurs, il peut être utilisé pour modifier les éléments du tableau. Bien que cela soit pratique et supporté, cela peut produire des résultats bizarres si LISTE n'est pas un tableau nommé. L'utilisation d'une boucle `foreach` est plus claire dans de nombreux cas. Voir aussi `grep` pour produire un tableau composé de tous les éléments de la liste originale pour lesquels BLOC ou EXPR est évalué à vrai (`true`).

**mkdir FILENAME,MASK****mkdir**

Crée le répertoire dont le nom est spécifié par FILENAME avec les droits d'accès spécifiés par MASK (et modifiés par `umask`). En cas de succès, retourne TRUE (vrai). Sinon, retourne false (faux) et positionne la variable `!` (`errno`). Par défaut, MASK vaut 0777.

En général, il vaut mieux créer des répertoires avec un MASK permissif et laisser l'utilisateur modifier cela via son `umask` que de fournir un MASK trop restrictif ne permettant pas à l'utilisateur d'être plus permissif. L'exception à cette règle concerne les répertoires ou les fichiers doivent être privés (fichiers de messagerie par exemple). `umask` discute plus en détails du choix de MASK.

Notez que, selon la norme POSIX 1003.1-1996, FILENAME peut se terminer par zéro, un ou plusieurs `\`. Certains systèmes d'exploitation ou de fichiers ne le permettent pas. Donc, pour que tout le monde soit content. Perl enlève automatiquement ces caractères `\` finaux.

**msgctl ID,CMD,ARG**

Appelle la fonction `msgctl(2)` des IPC System V. Vous devrez probablement dire :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si CMD est `IPC_STAT` alors ARG doit être une variable qui pourra contenir la structure `msqid_ds` retournée. Renvoie la même chose que `ioctl()` : la valeur undef en cas d'erreur, "0 but true" pour la valeur zéro ou la véritable valeur dans les autres cas. Voir aussi la documentation de `IPC::SysV` et `IPC::Semaphore::Msg`.

**msgget KEY,FLAGS**

Appelle la fonction `msgget(2)` des IPC System V. Retourne l'id de la queue de messages ou la valeur undef en cas d'erreur. Voir aussi `IPC::SysV` et `IPC::SysV::Msg`.

**msgrcv ID,VAR,SIZE,TYPE,FLAGS**

Appelle la fonction `msgrcv` des IPC System V pour recevoir un message à partir de la queue de message d'identificateur ID et le stocker dans la variable VAR avec une taille maximale de SIZE. Remarquez que, si un message est reçu, le type de message sera la première chose dans VAR et que la taille maximale de VAR est SIZE plus la taille du type de message. Ces données compactées peuvent être décompactées par `unpack("l! a*", $type)`. Souille la variable VAR. Retourne TRUE (vrai) en cas de succès ou false (faux) sinon. Voir aussi `IPC::SysV` et `IPC::SysV::Msg`.

**msgsnd ID,MSG,FLAGS**

Appelle la fonction `msgsnd` des IPC System V pour envoyer le message MSG dans la queue de messages d'identificateur ID. MSG doit commencer par l'entier long natif donnant le type de message suivi de la longueur réelle du message suivi du message lui-même. Cette donnée compactée peut être produite par `pack("l! a*", $type, $message)`. Retourne true (vrai) en cas de succès ou false (faux) en cas d'erreur. Voir aussi `IPC::SysV` et `IPC::SysV::Msg`.



**my** **EXPR****my** **TYPE** **EXPR****my** **EXPR** : **ATTRIBUTS****my** **TYPE** **EXPR** : **ATTRIBUTS**

`my()` déclare les variables listées comme étant locales (lexicalement) au bloc, fichier ou `eval()` englobant. Si plus d'une variable est listée, la liste doit être placée entre parenthèses.

La sémantique exacte et l'interface avec `TYPE` et `ATTRIBUTS` est encore en cours d'évolution. `TYPE` est actuellement lié à l'utilisation de la directive `fields` et les attributs sont gérés par la directive `attributes` ou, depuis la version 5.8.0 de Perl, via le module `Attribute::Handlers`. Voir `Variables Privées` via `my()` in *perlsb* pour plus de détails ainsi que *fields*, *attributes* et *Attribute::Handlers*.

**next** **LABEL****next**

La commande `next` fonctionne comme l'instruction `continue` du C ; elle commence la prochaine itération d'une boucle :

```
LINE: while (<STDIN>) {
 next LINE if /^#/; # ne pas traiter les commentaires
 #...
}
```

Si il y avait un bloc `continue` dans cet exemple, il serait exécuté même pour les lignes ignorées. Si `LABEL` est omis, la commande se réfère au bloc englobant le plus intérieur.

`next` ne peut pas être utilisé pour sortir d'un bloc qui doit retourner une valeur comme `eval {}`, `sub {}` ou `do {}` et ne devrait pas être utilisé pour sortir d'une opération `grep()` ou `map()`.

Voir aussi `continue` pour voir comment `last`, `next` et `redo` fonctionne.

**no** **Module** **VERSION** **LISTE****no** **Module** **VERSION****no** **Module** **LISTE****no** **Module**

Voir la fonction `use` dont `no` est le contraire.

**oct** **EXPR****oct**

Interprète `EXPR` comme une chaîne octale et retourne la valeur correspondante. (Si il s'avère que `EXPR` commence par `0x`, elle sera interprétée comme un chaîne hexadécimale. Si `EXPR` commence par `0b`, elle sera interprétée comme une chaîne binaire. Dans tous les cas, les blancs initiaux sont ignorés) La ligne suivante manipule les chaînes décimales, binaires, octales et hexadécimales comme le fait la notation standard Perl ou C :

```
$val = oct($val) if $val =~ /^0/;
```

Si `EXPR` est absent, la commande s'applique à `$_`. Pour réaliser l'opération inverse (produire la représentation octale d'un nombre), utilisez `sprintf()` ou `printf()` :

```
$perms = (stat("filename"))[2] & 07777;
$oct_perms = sprintf "%lo", $perms;
```

La fonction `oct()` est couramment utilisée pour convertir une chaîne telle que `644` en un mode d'accès pour fichier par exemple. (`perl` convertit automatiquement les chaînes en nombres si besoin mais en supposant qu'ils sont en base 10.)

**open** **DESCRIPTEUR,EXPR****open** **DESCRIPTEUR,MODE,EXPR****open** **DESCRIPTEUR,MODE,EXPR,LISTE****open** **DESCRIPTEUR,MODE,REFERENCE****open** **DESCRIPTEUR**

Ouvre le fichier dont le nom est donné par `EXPR` et l'associe à `DESCRIPTEUR`.

(La suite de cette section est une description exhaustive de `open()`. Pour une introduction plus douce, vous devriez regarder *perlopentut*.)

Si `DESCRIPTEUR` est une variable scalaire (ou un élément d'un tableau ou d'une table de hachage) indéfinie, cette variable recevra une référence vers un nouveau descripteur anonyme. Sinon, si `DESCRIPTEUR` est une expression,

sa valeur est utilisée en tant que nom du descripteur à associer. (Ceci est considéré comme une référence symbolique donc `use strict 'refs'` ne devrait *pas* être actif.)

Si `EXPR` est omis, la variable scalaire du même nom que le `DESCRIPTEUR` contient le nom du fichier. (Remarquez que les variables lexicales – celles déclarées par `my()` – ne fonctionnent pas dans ce cas ; donc si vous utilisez `my()`, spécifiez `EXPR` dans votre appel à `open`.)

Si au moins trois arguments sont spécifiés alors le mode d'ouverture et le nom du fichier sont séparés. Si `MODE` est '`<`' ou rien du tout, le fichier est ouvert en lecture. Si `MODE` est '`>`', le fichier est tronqué puis ouvert en écriture en étant créé si nécessaire. Si `MODE` est '`>>`', le fichier est ouvert en écriture et en mode ajout. Là encore, il sera créé si nécessaire.

Vous pouvez ajouter un '`+`' devant '`>`' ou '`<`' pour indiquer que vous voulez à la fois les droits d'écriture et de lecture sur le fichier ; Ceci étant '`+<`' est toujours mieux pour les mises à jour en lecture/écriture – le mode '`+>`' écraserait le fichier au préalable. Habituellement, il n'est pas possible d'utiliser le mode lecture/écriture pour des fichiers textes puisqu'ils ont des tailles d'enregistrements variables. Voir l'option `-i` dans `perlrun` pour une meilleure approche. Le fichier est créé avec les droits `0666` modifiés par la valeur de `umask` du processus courant.

Ces différents préfixes correspondent aux différents modes d'ouverture de `fopen(3)` : '`r`', '`r+`', '`w`', '`w+`', '`a`' et '`a+`'.

Dans sa forme à 1 ou 2 arguments, le mode et le nom de fichier peuvent être concaténés (dans cet ordre), éventuellement séparés par des espaces. Il est possible d'omettre le mode si c'est '`<`'.

Si le nom de fichier commence par '`|`', le nom de fichier est interprété comme une commande vers laquelle seront dirigées les sorties (via un tube – en anglais pipe) et si le nom de fichier se termine par '`|`', le nom de fichier est interprété comme une commande dont la sortie sera récupérée (via un tube – en anglais pipe). Voir Utilisation de `open()` pour la CIP in `perlipc` pour des exemples à ce sujet. (Vous ne pouvez pas utiliser `open()` pour une commande qui utiliserait un même tube à la fois pour ses entrées *et* pour ses sorties mais `IPC::Open2`, `IPC::Open3` et Communication Bidirectionnelle avec un autre Processus in `perlipc` proposent des solutions de remplacement.)

Pour les appels avec au moins trois arguments, si `MODE` est '`|-`', le nom de fichier est interprété comme une commande vers laquelle seront dirigées les sorties et si le `MODE` est '`-|`', le nom de fichier sera interprété comme une commande dont la sortie sera récupérée (via un tube – en anglais pipe). Pour retomber sur une forme à deux arguments, il suffit de remplacer le moins ('`-`') par la commande elle-même. Voir Utilisation de `open()` pour la CIP in `perlipc` pour des exemples à ce sujet. (Vous ne pouvez pas utiliser `open()` pour une commande qui utiliserait un même tube à la fois pour ses entrées *et* pour ses sorties mais `IPC::Open2`, `IPC::Open3` et Communication Bidirectionnelle avec un autre Processus in `perlipc` proposent des solutions de remplacement.)

Pour les appels avec au moins trois arguments et faisant au mécanisme de tube, si `LISTE` est spécifiée (ce sont les arguments après la commande) alors `LISTE` devient les arguments de la commande invoquée si la plateforme l'accepte. La sémantique de `open` avec plus de trois arguments pour un mode sans tube n'est pas encore spécifiée. Des "filtres" expérimentaux peuvent donner un sens à ces arguments supplémentaires.

Dans la forme à un ou deux arguments, ouvrir '`-`' revient à ouvrir `STDIN` tandis qu'ouvrir '`>-`' revient à ouvrir `STDOUT`.

Vous pouvez utiliser la forme à trois arguments en spécifiant des "filtres" d'entrée/sortie (anciennement appelés "disciplines") à appliquer au descripteur afin de modifier la manière dont sont traités les entrées et les sorties (Voir `open` et `PerlIO` pour plus d'informations). Par exemple :

```
open(FH, "<:utf8", "file")
```

ouvrira le fichier "file" encodé en UTF-8 et contenant des caractères Unicode. Voir `perluniintro`. Remarquez que si des filtres sont spécifiés dans un appel utilisant la syntaxe à trois arguments alors les filtres par défaut fixés stockés dans `$_OPEN` (voir `perlvar` ; habituellement fixés par la directive `open` ou par les options `-CioD`) sont ignorés.

`Open` renvoie une valeur non nulle en cas de succès et `undef` sinon. Si `open()` utilise un tube, la valeur de retour sera le PID du sous-processus.

Si vous utilisez Perl sur un système qui fait une distinction entre les fichiers textes et les fichiers binaires alors vous devriez regarder du côté de `binmode()` pour connaître les astuces à ce sujet. La principale différence entre les systèmes nécessitant `binmode()` et les autres réside dans le format de leurs fichiers textes. Des systèmes tels que Unix, MacOS et Plan9 qui délimitent leurs lignes par un seul caractère et qui encodent ce caractère en C par "`\n`" ne nécessitent pas `binmode()`. Les autres en ont besoin.

À l'ouverture d'un fichier, c'est généralement une mauvaise idée de continuer l'exécution normale si la requête échoue, ce qui explique pourquoi `open()` est si fréquemment utilisé en association avec `die()`. Même si `die()` n'est pas ce que vous voulez faire (par exemple dans un script CGI où vous voulez récupérer un message d'erreur joliment présenté (il y a des modules qui peuvent vous aider pour cela)), vous devriez toujours vérifier la valeur retournée par l'ouverture du fichier. L'une des rares exceptions est lorsque vous voulez travailler sur un descripteur non ouvert.

Un appel sous la forme de trois arguments en mode lecture/écriture avec un troisième argument indéfini (`undef`) est un cas spécial :

```
open(TMP, "+>", undef) or die ...
```

ouvrira un descripteur vers un fichier temporaire anonyme. Par souci de symétrie, cela marche aussi avec le mode "+<" mais il sera quand même plus pratique d'écrire quelque chose d'abord dans le fichier. Vous aurez besoin de faire appel à seek() avant de pouvoir lire le contenu du fichier.

Depuis la version 5.8.0, par défaut lors de sa compilation, perl est configuré pour utiliser PerlIO. Si vous n'avez pas touché à cela (ex. Configure -Uuseperlio), les descripteurs peuvent être ouverts sur des fichiers "en mémoire" stockés dans des scalaires Perl via :

```
open($fh, '>', \ $variable) || ...
```

Par contre, si vous essayez de ré-ouvrir STDOUT ou STDERR comme des fichiers "en mémoire", vous devez les fermer auparavant :

```
close STDOUT;
open STDOUT, '>', \ $variable or die "Can't open STDOUT: $!";
```

Exemples :

```
$ARTICLE = 100;
open ARTICLE or die "Can't find article $ARTICLE: $!\n";
while (<ARTICLE>) {...

open(LOG, '>>/usr/spool/news/twitlog'); # (log is reserved)
si le open échoue, les sorties sont perdues

open(DBASE, '+<', 'dbase.mine') # ouverture pour mise à jour
 or die "Can't open 'dbase.mine' for update: $!";

open(DBASE, '+<dbase.mine') # idem
 or die "Can't open 'dbase.mine' for update: $!";

open(ARTICLE, '-|', "caesar <$article") # décryptage de l'article
 or die "Can't start caesar: $!";

open(ARTICLE, "caesar <$article |") # idem
 or die "Can't start caesar: $!";

open(EXTRACT, "|sort >Tmp$$") # $$ est notre ID de process
 or die "Can't start sort: $!";

fichiers en mémoire
open(MEMORY, '>', \ $var)
 or die "Can't open memory file: $!";
print MEMORY "foo!\n"; # les sorties seront envoyées vers $var

traitement de la liste des fichiers fournie en argument
foreach $file (@ARGV) {
 process($file, 'fh00');
}

sub process {
 my($filename, $input) = @_;
 $input++; # c'est une incrémentation de chaîne
 unless (open($input, $filename)) {
 print STDERR "Can't open $filename: $!\n";
 return;
 }

 local $_;
 while (<$input>) { # remarquez l'utilisation de l'indirection
 if (/^#include "(.*)"/) {
 process($1, $input);
 next;
 }
 #... # ce qu'il faut faire...
 }
}
```

Voir *perliol* pour plus de détails sur PerlIO.

Vous pouvez aussi, dans la tradition du Bourne shell, spécifier une expression *EXPR* commençant par '>&' auquel cas le reste de la chaîne sera interprété comme le nom d'un descripteur (ou son numéro si c'est une valeur numérique) à dupliquer puis à ouvrir. Vous pouvez utiliser & après >, >>, <, +>, +>> et +<. Le mode que vous spécifiez devrait correspondre à celui du descripteur original. (La duplication d'un descripteur ne prend pas en compte l'éventuel contenu des tampons d'entrées/sorties). Si vous utilisez l'appel avec 3 arguments alors vous devez fournir soit un nombre, soit le nom du descripteur soit une référence globale "normale".

Voici un script qui sauvegarde, redirige et restaure STDOUT et STDERR de différentes manières :

```
#!/usr/bin/perl
open my $oldout, ">&STDOUT" or die "Can't dup STDOUT: $!";
open OLDERR, ">&", *STDERR or die "Can't dup STDERR: $!";

open STDOUT, '>', "foo.out" or die "Can't redirect STDOUT: $!";
open STDERR, ">&STDOUT" or die "Can't dup STDOUT: $!";

select STDERR; $| = 1; # make unbuffered
select STDOUT; $| = 1; # make unbuffered

print STDOUT "stdout 1\n"; # this works for
print STDERR "stderr 1\n"; # subprocesses too

open STDOUT, ">&", $oldout or die "Can't dup \$oldout: $!";
open STDERR, ">&OLDERR" or die "Can't dup OLDERR: $!";

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

Si vous spécifiez '<&=N', où N est un numéro de descripteur ou un descripteur de fichier, alors Perl fera l'équivalent d'un *fdopen()* en C sur ce descripteur de fichier (sans faire appel à *dup(2)*); c'est plus économe en descripteur de fichier. Par exemple :

```
ouverture en lecture, en réutilisant le numéro de $fd
open(DESCRIPTEUR, "<&=$fd")

ou

open(DESCRIPTEUR, "<&=", $fd)

ou

ouverture pour ajout, en réutilisant le numéro de OLDFH
open(FH, ">>&=", OLDFH)

ou

open(FH, ">>&=OLDFH")
```

Au-delà de l'économie elle-même, cela est utile si vous avez des choses liées à ce descripteur, comme par exemple un verrou posé via *flock()*. Si vous faites juste appel à *open(A, '>>B')*, le descripteur A n'est pas le même que B et donc un *flock(A)* ne fera pas un *flock(B)*, et vice-versa. Alors qu'avec *open(A, '>>&=B')*, A et B se partagent le même descripteur.

Remarquez que, si vous utilisez une version de Perl antérieure à la version 5.8.0, cette fonctionnalité dépend de la fonction *fdopen()* de votre bibliothèque C. Sur la plupart des systèmes UNIX, *fdopen()* est connu pour échouer lorsque le nombre de descripteurs de fichiers dépasse une certaine limite, typiquement 255. Avec Perl 5.8.0 et au-delà, PerlIO est adopté par défaut et ne pose pas cette limite.

Vous pouvez savoir si Perl a été compilé avec PerlIO ou non en exécutant *perl -V* et en regardant la ligne *useperlio=*. Si *useperlio* est *define*, vous avez PerlIO.

Si vous ouvrez un tube (pipe) sur la commande '--', i.e. soit '|-' soit '|', alors vous faites un fork implicite et la valeur de retour de *open* est le PID du processus fils pour le processus père et 0 pour le processus fils. (Utilisez *defined(\$pid)* pour savoir si le *open* s'est bien déroulé.) Le descripteur se comporte normalement pour le père mais pour le processus fils, les entrées/sorties sur ce descripteur se font via STDIN/STDOUT. Dans le processus fils, le descripteur n'est pas ouvert – les échanges se font via les nouveaux STDIN ou STDOUT. Cela s'avère utile lorsque vous voulez avoir un contrôle plus fin sur la commande exécutée par exemple lorsque vous avez un script *setuid* et que vous ne voulez pas que le shell traite les méta-caractères dans les commandes. Les lignes suivantes sont quasiment équivalentes (trois à trois) :

```

open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, '|-', "tr '[a-z]' '[A-Z]'");
open(FOO, "|-" || exec 'tr', '[a-z]', '[A-Z]');
open(FOO, '|-', "tr", '[a-z]', '[A-Z]');

open(FOO, "cat -n '$file'|");
open(FOO, '-|', "cat -n '$file'");
open(FOO, "-|" || exec 'cat', '-n', $file);
open(FOO, '-|', "cat", '-n', $file);

```

Le dernier exemple de chacun des deux blocs montre l'utilisation des tubes avec le passage de paramètres. Ce n'est pas disponible sur toutes les plateformes. Si votre plateforme propose un vrai `fork()` (en d'autres termes, si vous êtes sur UNIX) alors vous pouvez utiliser cette syntaxe.

Voir Ouvertures Sûres d'un Tube in *perlipc* pour plus d'exemples à ce sujet.

À partir de la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant toute opération impliquant un `fork` mais ce n'est pas supporté sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appelé la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts afin d'éviter toute perte de données.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (`close-on-exec`) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de `$^F`. Voir `$^F` in *perlvar*.

La fermeture d'un descripteur utilisant un tube (pipe) amène le processus père à attendre que son fils se termine puis à retourner la valeur de statut dans  `$?` .

Le nom de fichier passé à `open` dans la forme à 1 ou 2 arguments verra ses éventuels espaces avant et après supprimés et les caractères de redirection normaux seront respectés. Cette fonctionnalité, connue sous le nom de "ouverture magique", autorise plein de bonnes choses. Un utilisateur peut spécifier un nom de fichier tel que `"rsh cat file |"` ou alors vous pouvez modifier certains noms de fichiers selon vos besoins :

```

$filename =~ s/(.*\.gz)\s*/gzip -dc < $1|/;
open(FH, $filename) or die "Can't open $filename: $!";

```

Utilisez la forme à 3 arguments pour ouvrir un fichier dont le nom contient des caractères quelconques :

```
open(FOO, '<', $file);
```

sinon il est nécessaire de protéger les caractères spéciaux et tous les espaces avant et/ou après :

```

$file =~ s/^(\\s)#./$1#;
open(FOO, "< $file\0");

```

(cela peut ne pas fonctionner sur certains systèmes de fichier bizarres). Vous devez choisir consciemment entre la forme magique ou la forme à 3 arguments de `open()` :

```
open IN, $ARGV[0];
```

autorisera l'utilisateur à spécifier un argument de la forme `"rsh cat file |"` mais ne marchera pas avec un nom de fichier contenant des espaces alors que :

```
open IN, '<', $ARGV[0];
```

a exactement les limitation inverses.

Si vous voulez un "vrai" `open()` à la C (voir *open(2)* sur votre système) alors vous devriez utiliser la fonction `sysopen()` qui ne fait rien de magique. C'est un autre moyen de protéger vos noms de fichiers de toute interprétation. Par exemple :

```

use IO::Handle;
sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
 or die "sysopen $path: $!";
$oldfh = select(HANDLE); $| = 1; select($oldfh);
print HANDLE "stuff $$\n";
seek(HANDLE, 0, 0);
print "File contains: ", <HANDLE>;

```

En utilisant le constructeur du package `IO::Handle` (ou de l'une de ses sous-classes telles que `IO::File` ou `IO::Socket`), vous pouvez générer des descripteurs anonymes qui ont la même portée que les variables qui gardent une référence sur eux et qui se ferment automatiquement dès qu'ils sont hors de portée :

```

use IO::File;
#...
sub read_myfile_munged {
 my $ALL = shift;
 my $handle = new IO::File;
 open($handle, "myfile") or die "myfile: $!";
 $first = <$handle>
 or return (); # Fermeture automatique ici
 mung $first or die "mung failed"; # Ou ici.
 return $first, <$handle> if $ALL; # Ou ici.
 $first; # Ou ici.
}

```

Voir `seek()` pour de plus amples informations sur le mélange entre lecture et écriture.

### **opendir DIRHANDLE,EXPR**

Ouvre le répertoire nommé par `EXPR` pour un traitement par `readdir()`, `telldir()`, `seekdir()`, `rewinddir()` et `closedir()`. Retourne `true` (vrai) en cas de succès. `DIRHANDLE` peut être une expression dont la valeur sera utilisé comme `dirhandle` indirect, habituellement le nom du vrai `dirhandle`. Si `DIRHANDLE` est une variable scalaire (ou un élément d'un tableau ou d'une table de hachage) dont la valeur n'est pas définie, cette variable recevra la référence au nouveau `dirhandle` anonyme. Les `DIRHANDLES` ont leur propre espace de noms séparé de celui des `DESCRIPTEURS`.

### **ord EXPR**

#### **ord**

Retourne la valeur numérique (dans l'encodage 8-bit natif comme ASCII, EBCDIC ou Unicode) du premier caractère de `EXPR`. Si `EXPR` est `omis`, utilise `$_`.

Voir `chr` pour l'opération inverse. Voir *perlunicode* et *encoding* pour en savoir plus sur Unicode.

### **our EXPR**

#### **our TYPE EXPR**

#### **our EXPR : ATTRS**

#### **our TYPE EXPR : ATTRS**

`our` permet d'associer un nom simple avec une variable du package courant utilisable dans la portée courante. Lorsque `use strict 'vars'` est actif, `our` vous permet d'utiliser des variables globales sans les qualifier complètement avec leur nom de package, dans la portée de la déclaration `our`. En cela, `our` diffère de `use vars` qui a une portée lié au package.

Au contraire de `my` qui alloue un espace de stockage et, dans la portée courante, associe un nom simple à cet espace, `our` associe un nom simple à une variable du package courant, dans la portée courante. En d'autres termes, `our` a la même portée que `my` mais ne crée pas nécessairement la variable associée.

Si plusieurs noms sont listées, la liste doit être placée entre parenthèses.

```

our $foo;
our ($fbar, $baz);

```

Un `our` déclare les variables listées comme étant des variables globales valides dans la portée d'un bloc englobant, d'un fichier ou d'un `eval`. Il a donc la même portée qu'une déclaration `"my"` mais ne crée pas de variable locale. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses. La déclaration `our` n'a aucun effet sémantique à moins que `"use strict vars"` soit actif auquel cas, cela vous permet d'utiliser les variables globales déclarées sans les qualifier par un nom de package. (Mais seulement dans la portée lexicale de votre déclaration `our`. En cela, il diffère de `"use vars"` dont la portée est globale au package.)

Un déclaration `our` déclare un variable globale qui sera visible dans toute la portée lexicale de cette déclaration, même entre différents packages. Le package d'appartenance de la variable est déterminée lors de la déclaration et non lors de l'utilisation. Cela signifie que vous aurez les comportements suivants :

```

package Foo;
our $bar; # déclare $Foo::bar
$bar = 20;

package Bar;
print $bar; # affiche 20 (valeur de $Foo::bar)

```

De multiples déclarations `our` sont autorisées dans la même portée lexicale si elles sont dans des packages différents. Si elles sont dans le même package, Perl produira un avertissement si vous le lui avez demandé, exactement comme pour de multiples déclarations `my`. Par contre, à l'inverse d'une deuxième déclaration `my` qui sera associée à une nouvelle variable, une deuxième déclaration `our` dans le même package et dans la même portée sera redondante.

```
use warnings;
package Foo;
our $bar; # déclare $Foo::bar
$bar = 20;

package Bar;
our $bar = 30; # déclare $Bar::bar
print $bar; # affiche 30

our $bar; # émission d'un avertissement
 # mais sans autre effet
print $bar; # affiche encore 30
```

Une déclaration `our` peut aussi être associée à une liste d'attributs.

La sémantique exacte et l'interface de `TYPE` et `ATTRS` sont encore en cours d'évolution. `TYPE` est actuellement lié à l'utilisation de la directive `fields` et les attributs (`ATTRS`) sont gérés par la directive `attributes` ou, depuis perl 5.8.0, via le module `Attribute::Handlers`. Voir *Variables Privées via my() in perlsub* pour plus de détails ainsi que *fields*, *attributes* et *Attribute::Handlers*.

Actuellement, le seul attribut reconnu par `our()` est `unique` qui indique qu'une seule et unique instance de la variable globale doit être utilisée par tous les interpréteurs que lance le programme dans un environnement d'interpréteurs multiples. (Le comportement par défaut est que chaque interpréteur possède sa propre instance.) Exemples :

```
our @EXPORT : unique = qw(foo);
our %EXPORT_TAGS : unique = (bar => [qw(aa bb cc)]);
our $VERSION : unique = "1.00";
```

Notez aussi que cet attribut a pour effet de rendre la variable globale non-modifiable dès que le premier interpréteur est cloné (par exemple lorsque le premier fil d'exécution est créé).

Un environnement d'interpréteurs multiples peut aussi provenir de l'émulation de `fork()` sur les plateformes Windows ou d'un interpréteur embarqué dans une application multi-fils (multi-threaded application). L'attribut `unique` ne fait rien dans tout autre environnement.

Attention : l'implémentation actuelle de cet attribut opère sur le typeglob associé à la variable ; cela signifie que `our $x : unique` a aussi pour effet `our @x : unique`; `our %x : unique`. Cela pourrait changer.

### pack TEMPLATE,LISTE

Prend une liste de valeurs et les transforme en une chaîne en utilisant les règles décrites par `TEMPLATE`. La chaîne résultante est la concaténation des valeurs converties. Typiquement, chaque valeur convertie aura sa représentation du niveau machine. Par exemple, sur une machine 32-bit, un entier converti sera représenté par une séquence de 4 octets.

Le `TEMPLATE` est une séquence de caractères qui donne l'ordre et le type des valeurs, de la manière suivante :

- a Une chaîne contenant des données binaires quelconques, éventuellement complétée par des caractères NUL.
- A Un texte (ASCII), complétée par des blancs.
- Z Une chaîne (ASCII) terminée par un caractère NUL, complétée par des caractères NUL.
- b Une chaîne de bits (en ordre croissant dans chaque octet, comme pour `vec()`).
- B Une chaîne de bits (en ordre décroissant dans chaque octet).
- h Une chaîne hexadécimale (chiffres hexadécimaux faibles en premier).
- H Une chaîne hexadécimale (chiffres hexadécimaux forts en premier).
- c La valeur d'un caractère signé.
- C La valeur d'un caractère non signé. Ne traite que des octets. Voir U pour Unicode.
- s La valeur d'un entier court (short) signé
- S La valeur d'un entier court (short) non signé  
(Ce 'short' est `_exactement_` sur 16 bits ce qui peut

- être différent de ce qu'un compilateur C local appelle 'short'. Si vous voulez un short natif, utilisez le suffixe '!'.)
- i La valeur d'un entier (integer) signé.
  - I La valeur d'un entier (integer) non signé.  
(Cet 'integer' est `_au moins_` sur 32 bits. Sa taille exact dépend de ce que le compilateur C local appelle 'int' et peut même être plus long que le 'long' décrit à l'item suivant.)
  - l La valeur d'un entier long (long) signé.
  - L La valeur d'un entier long (long) non signé.  
(Ce 'long' est `_exactement_` sur 32 bits ce qui peut être différent de ce qu'un compilateur C local appelle 'long'. Si vous voulez un long natif, utilisez le suffixe '!'.)
  - n Un entier court non signé (short) dans l'ordre "réseau" (big-endian).
  - N Un entier long non signé (long) dans l'ordre "réseau" (big-endian).
  - v Un entier court non signé (short) dans l'ordre "VAX" (little-endian).
  - V Un entier long non signé (long) dans l'ordre "VAX" (little-endian).  
(Ces 'shorts' et ces 'longs' font `_exactement_` 16 et 32 bits respectivement.)
  - q Une valeur quad (64-bit) signée.
  - Q Une valeur quad non signée.  
(Les quads ne sont disponibles que si votre système supporte les entiers 64-bit `_et_` que si Perl a été compilé pour les accepter. Provoquera une erreur fatale sinon.)
  - j Une valeur entière signée (un entier interne de Perl, IV)
  - J Une valeur entière non-signée (un entier non-signé de Perl, UV)
  - f Un flottant simple précision au format natif.
  - d Un flottant double précision au format natif.
  - F Une valeur à virgule flottante en format natif  
(un flottant Perl interne, NV).
  - D Un flottant long en double précision en format natif  
(Ce format doit être connue de votre système `_ET_` Perl doit être compilé pour les reconnaître.  
Produit une erreur fatale sinon.)
  - p Un pointeur vers une chaîne terminée par un caractère NUL.
  - P Un pointeur vers une structure (une chaîne de longueur fixe).
  - u Une chaîne uuencodée.
  - U Un nombre d'un caractère Unicode. Encode en UTF-8 en interne  
(ou en UTF-EBCDIC sur les plateformes EBCDIC).
  - w Un entier BER compressé (mais pas un ASN.1 BER, voir `perlpacktut` pour les détails). Ses octets représentent chacun un entier non signé en base 128. Les chiffres les plus significatifs viennent en premier et il y a le moins de chiffres possibles. Le huitième bit (le bit de poids fort) est toujours à 1 pour chaque octets sauf le dernier.
  - x Un octet nul.
  - X Retour en arrière d'un octet.
  - @ Remplissage par des octets nuls jusqu'à une position absolue,  
comptée depuis le début du groupe () englobant le plus proche.
  - ( Début d'un ()-groupe.

Les règles suivantes s'appliquent :



- Chaque lettre peut éventuellement être suivie d'un nombre spécifiant le nombre de répétitions. Pour tous les types sauf a, A, Z, b, B, h, H, @, x, X et P, la fonction pack utilisera autant de valeurs de la liste LISTE que nécessaire. Une \* comme valeur de répétition demande à utiliser toutes les valeurs restantes excepté pour @, x et X où c'est équivalent à 0 et pour u où c'est équivalent à 1 (ou 45, ce qui est la même chose). Le nombre de répétitions peut éventuellement être entouré de crochets comme dans pack 'C[80]', @arr.  
Il est possible de remplacer ce nombre de répétitions par un template placé entre crochets. C'est alors la longueur en octets de ce template compacté qui est utilisée comme nombre de répétitions. Par exemple, x[L] sautera autant d'octets que le nombre d'octets composant un long ; Le template \$x X[\$t] \$t pourra décompacter deux fois ce que décompactera \$t. Si le template entre crochets contient des commandes d'alignement (comme dans x![d]), sa longueur compactée sera calculée en supposant que le début du template est à l'alignement maximal. Utilisé avec Z, \* déclenchera l'ajout d'un octet nul final (donc la valeur compactée sera d'un octet plus longue que la longueur (length) de la valeur initiale). La valeur de répétition pour u est interprétée comme le nombre maximale d'octets à encoder par ligne produite avec 0 et 1 remplacé par 45.
- Les types a, A et Z n'utilisent qu'une seule valeur mais la compacte sur la longueur spécifiée en la complétant éventuellement par des espaces ou des caractères NUL. Lors du décompactage, A supprime les espaces et les caractères NUL finaux, Z supprime tout ce qui suit le premier caractère NUL alors que a laisse la valeur complète. Lors du compactage, a et Z sont équivalents.  
Si la valeur à compacter est trop longue, elle est tronquée. Si elle est trop longue et qu'une longueur explicite \$count est fournie, Z compactera uniquement \$count-1 octets suivi d'un octet nul. Donc, Z compacte un caractère nul final en toutes circonstances.
- De la même manière, les types b et B remplissent la chaîne compactée avec autant de bits que demandés. Chaque octet du champ d'entrée de pack() produira 1 bit dans le résultat. Chaque bit résultant est le bit le moins significatif de l'octet d'entrée (i.e. ord(\$octet)%2). En particulier, les octets "0" et "1" produisent les bits 0 et 1 exactement comme le font "\0" et "\1".  
En partant du début de la chaîne d'entrée de pack(), chaque 8-uplet d'octets est converti en un octet de sortie. Avec le format b, le premier octet du 8-uplet détermine le bit le moins significatif de l'octet alors qu'avec le format B, il détermine le bit le plus significatif.  
Si la longueur de la chaîne d'entrée n'est pas exactement divisible par 8, le reste est compacté comme si la chaîne d'entrée était complétée par des octets nuls à la fin. De manière similaire, lors du décompactage, les bits "supplémentaires" sont ignorés.  
Si la chaîne d'entrée de pack() est plus longue que nécessaire, les octets en trop sont ignorés. Une valeur de répétition de \* demande à utiliser tous les octets du champ d'entrée. Lors du décompactage, les bits sont convertis en une chaîne de "0" et de "1".
- h et H compactent une chaîne hexadécimale (par groupe de 4-bit représentant un chiffre hexadécimale 0-9a-f). Chaque octet du champ d'entrée de pack() génère 4 bits du résultat. Pour les octets non alphabétiques, le résultat est basé sur les 4 bits les moins significatifs de l'octet considéré (i.e. sur ord(\$octet)%16). En particulier, les octets "0" et "1" génèrent 0 et 1 comme le font les octets "\0" et "\1". Pour les octets "a".."f" et "A".."F", le résultat est compatible avec les chiffres hexadécimaux habituels et donc "a" et "A" génèrent tous les deux le groupe de 4 bits 0xa==10. Le résultat pour les octets "g".."z" et "G".."Z" n'est pas clairement défini.  
En partant du début de la chaîne d'entrée de pack(), chaque paire d'octets est convertie en 1 octet de sortie. Avec le format h, le premier octet de la paire détermine les 4 bits les moins significatifs de l'octet résultant alors qu'avec le format H, il détermine les 4 bits les plus significatifs.  
Si la longueur de la chaîne d'entrée n'est pas paire, pack() se comportera comme si un octet nul avait été ajouté à la fin. De manière similaire, lors du décompactage (par unpack()), les groupes de 4 bits supplémentaires sont ignorés.  
Si la chaîne d'entrée de pack() est plus longue que nécessaire, les octets supplémentaires sont ignorés. Une valeur de répétition de \* demande à utiliser tous les octets de la chaîne d'entrée. Lors du décompactage, les bits sont convertis en une chaîne de chiffres hexadécimaux.
- Le type "p" compactera une chaîne terminée par un caractère NUL dont on donne le pointeur. Il est de votre responsabilité de vous assurer que la chaîne n'est pas une valeur temporaire (qui pourrait être désallouée avant l'utilisation de la valeur compactée). Le type P compactera une structure de la taille indiquée par la longueur. Un pointeur NULL est créé si la valeur correspondant à "p" ou "P" est undef. Idem pour unpack().
- Le caractère / dans TEMPLATE autorise le compactage et le décompactage de chaîne donc la représentation compacte est la longueur en octets suivie de la chaîne elle-même. Vous écrivez longueur-item/chaîne-item.  
La longueur-item peut être n'importe quelle lettre d'un template pack et elle représente comment cette valeur de longueur est compactée. Les plus utilisées sont les longueurs entières comme n (pour des chaînes Java), w (pour ASN.1 ou SNMP) et N (pour les XDR de Sun).  
Pour le compactage (via pack), chaîne-item doit être, s'il est présent, "A\*", "a\*" ou "Z\*". Pour le décompactage (via unpack), la longueur de la chaîne est celle obtenue par longueur-item mais si vous spécifiez '\*', elle sera ignorée. Pour tout autre code, unpack applique la valeur de longueur à l'item suivant qui ne doit pas avoir de

compteur de répétition.

```
unpack 'C/a', "\04Gurusamy"; donne 'Guru'
unpack 'a3/A* A*', '007 Bond J '; donne (' Bond','J')
pack 'n/a* w/a*', 'hello,','world'; donne "\000\006hello,\005world"
```

*longueur-item* n'est pas retourné explicitement par `unpack()`.

L'ajout d'un compteur de répétition à la lettre *longueur-item* est inutile sauf si cette lettre est A, a ou Z. Le compactage avec une *longueur-item* spécifiée par a ou Z peut introduire des caractères "\000" que Perl ne considérera pas comme légal dans une chaîne numérique.

- Les types entiers s, S, l et L peuvent être immédiatement suivi d'un suffixe ! pour indiquer des shorts ou des longs natifs – comme vous pourrez le voir dans l'exemple suivant, l ne signifie pas exactement 32 bits puisque le long natif (tel qu'il est vu par le compilateur C natif) peut être plus long. C'est une préoccupation principalement sur des plates-formes 64-bit. Vous pouvez voir si l'utilisation de ! fait une différence en faisant :

```
print length(pack("s")), " ", length(pack("s!")), "\n";
print length(pack("l")), " ", length(pack("l!")), "\n";
```

i! et I! fonctionne aussi mais uniquement dans un souci de complétude puisqu'ils sont totalement identique à i et I.

La taille réelle (en octets) des short, int, long et long long natifs sur la plate-forme où Perl a été installé est aussi disponible via *Config* :

```
use Config;
print $Config{shortsize}, "\n";
print $Config{intsize}, "\n";
print $Config{longsize}, "\n";
print $Config{longlongsize}, "\n";
```

(\$Config{longlongsize} sera indéfini (undef) si votre système ne supporte pas les long long).

- Les formats entiers s, S, i, I, l, L, j et J sont par construction non portables entre processeurs et entre systèmes d'exploitation puisqu'ils respectent l'ordre (big-endian ou little-endian) natif. Par exemple l'entier sur 4 octets 0x12345678 (305419896 en décimal) devrait être nativement ordonné (stocké et manipulé par les registres de la CPU) en octets comme :

```
0x12 0x34 0x56 0x78 # ordre little-endian
0x78 0x56 0x34 0x12 # ordre big-endian
```

À la base, les CPU des familles Intel et VAX sont little-endian alors que tous les autres, par exemple Motorola m68k/88k, PPC, Sparc, HP PA, Power et Cray, sont big-endian. Les puces Alpha et MIPS peuvent être les deux : Digital les utilise(aient) en mode little-endian ; SGI/Cray les utilise en mode big-endian.

Les noms "big-endian" et "little-endian" sont des références au grand classique "Les voyages de Gulliver" (au travers de l'article "On Holy Wars and a Plea for Peace" par Danny Cohen, USC/ISI IEN 137, April 1, 1980) et donc aux habitudes des mangeurs d'oeufs Lilliputiens.

Quelques systèmes peuvent même avoir un ordre des octets encore plus bizarre tel que :

```
0x56 0x78 0x12 0x34
0x34 0x12 0x78 0x56
```

Vous pouvez voir les préférences de votre systèmes par

```
print join(" ", map { sprintf "%#02x", $_ }
 unpack("C*",pack("L",0x12345678))), "\n";
```

L'ordre des octets de la plate-forme où Perl a été installé est aussi disponible via *Config* :

```
use Config;
print $Config{byteorder}, "\n";
```

Les ordres d'octets '1234' et '12345678' sont little-endian alors que '4321' et '87654321' sont big-endian.

Si vous voulez des entiers compactés portables, il vous faut utiliser les formats n, N, v et V puisque leur taille et l'ordre de leurs octets sont connus. Voir aussi *perlport*.

- Les nombres réels (simple et double précision) sont uniquement au format natif de la machine. À cause de la multiplicité des formats existants pour les flottants et du manque d'une représentation "réseau" standard, aucune possibilité d'échange n'est proposée. Cela signifie que les données réelles compressées sur une machine peuvent ne pas être lisibles par une autre machine – même si elles utilisent toutes deux l'arithmétique flottante IEEE (puisque l'ordre des octets de la représentation mémoire ne fait pas partie des spécifications IEEE). Voir aussi *perlport*.

Sachez que Perl utilise des doubles en interne pour tous les calculs numériques et qu'une conversion de double vers float puis retour vers double amène à une perte de précision (i.e., `unpack("f", pack("f", $foo))` est généralement différent de `$foo`).

- Si le template débute par un U, la chaîne résultat sera traité comme de l'Unicode codé en UTF-8. Vous pouvez forcer l'encodage UTF-8 dans une chaîne en la commençant par U0 et les octets qui suivront seront interprétés comme des caractères Unicode. Si vous ne voulez pas que cela arrive, vous pouvez commencer votre template par C0 (ou autre chose) pour forcer Perl à ne pas coder votre chaîne en UTF-8 puis le faire suivre d'un U\* plus

- tard.
- Vous devez réaliser vous-même tous les alignements et les remplissages nécessaires en insérant par exemple assez de `x` lors du compactage. `pack()` et `unpack()` n'ont aucun moyen de savoir d'où viennent et où iront les octets. Par conséquent, `pack()` (et `unpack()`) gère leurs entrée et leurs sortie comme des séquences d'octets à plat.
  - Un `()`-groupe est un sous-TEMPLATE entouré de parenthèses. Un tel groupe peut être répété soit en le postfixant par un compteur de répétition soit, pour `unpack()`, en le préfixant par un compteur via le caractère `/`. Lors de chaque répétition d'un groupe, le positionnement via `@` recommence à zéro. Si bien que le résultat de :
 

```
pack('@1A(@2A)@3A', 'a', 'b', 'c')
```

 est la chaîne `"\0a\0\0bc"`.
  - `x` et `X` accepte le modificateur `!`. Dans ce cas, ils agissent comme des commandes d'alignement : ils sautent vers la position la plus proche alignée sur un multiple de `count` octets. Par exemple, pour compacter ou décompacter la structure C `struct {char c; double d; char cc[2]}`, on peut utiliser le TEMPLATE `C x![d] d C[2]` ; Cela garantit que les doubles seront alignés sur la taille d'un double. Pour les commandes d'alignement un `count` de 0 est équivalent à un `count` de 1 ; Et tous les deux sont sans effet.
  - Dans TEMPLATE, un commentaire commence par `#` et se termine en fin de ligne. Vous pouvez utiliser de blancs pour séparer les codes de compactage les uns des autres, mais le modificateur `!` ou le compteur de répétition doivent suivre immédiatement un code.
  - Si TEMPLATE nécessite plus d'argument que ceux réellement fournis, `pack()` supposera que des arguments "" supplémentaires sont fournis. Si TEMPLATE nécessite moins d'arguments que ceux réellement fournis à `pack()`, les arguments en trop sont ignorés.

Exemples :

```
$foo = pack("CCCC",65,66,67,68);
foo eq "ABCD"
$foo = pack("C4",65,66,67,68);
même chose
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
même chose avec des lettres cerclées Unicode

$foo = pack("ccxxcc",65,66,67,68);
foo eq "AB\0\0CD"

note : les exemples précédents utilisant "C" et "c" ne sont
corrects que sus des systèmes ACSII ou dérivés comme ISO Latin 1
ou UTF-8. En EBCDIC, le premier exemple devrait être
$foo = pack("CCCC",193,194,195,196);

$foo = pack("s2",1,2);
"\1\0\2\0" sur little-endian
"\0\1\0\2" sur big-endian

$foo = pack("a4","abcd","x","y","z");
"abcd"

$foo = pack("aaaa","abcd","x","y","z");
"axyz"

$foo = pack("a14","abcdefg");
"abcdefg\0\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
une vraie struct tm (sur mon système en tous cas)

$utmp_template = "Z8 Z8 Z16 L";
$utmp = pack($utmp_template, @utmp1);
une struct utmp (BSD-isme)

@utmp2 = unpack($utmp_template, $utmp);
"@utmp1" eq "@utmp2"

sub bintodec {
 unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

```
$foo = pack('sx2l', 12, 34);
short 12, two zero bytes padding, long 34
$bar = pack('s@4l', 12, 34);
short 12, zero fill to position 4, long 34
$foo eq $bar
```

La même valeur de `TEMPLATE` peut généralement être utilisée avec la fonction `unpack()`.

### package

#### package NAMESPACE

Déclare l'unité de compilation comme faisant partie de l'espace de nommage `NAMESPACE`. La portée de la déclaration `package` (N.d.t : paquetage en français) commence à la déclaration elle-même et se termine à la fin du bloc, du fichier ou de l'éval englobant (c'est la même portée que l'opérateur `local()`). Tout identificateur dynamique non qualifié à venir sera dans cet espace de nommage. Une instruction `package` n'affecte que les variables dynamiques – même celles sur lesquelles vous utilisez `local()` – et *non* les variables lexicales créées par `my()`. C'est typiquement la première déclaration dans un fichier inclus par les opérateurs `require` ou `use`. Vous pouvez basculer vers un même package en plusieurs endroits ; cela ne fait que changer la table de symboles utilisée par le compilateur pour la suite du bloc. Vous pouvez référencer des variables ou des descripteurs de fichiers d'autres packages en préfixant l'identificateur par le nom du package suivi de deux fois deux points : `$Package::Variable`. Si le nom de package est vide, c'est la package `main` qui est utilisé. Donc `$$::sail` est équivalent à `$main::sail` (et aussi à `$main'sail` qui peut encore se voir dans du vieux code)..

Si `NAMESPACE` est omis alors il n'y a pas de package courant et tous les identificateurs doivent être soit complètement qualifiés soit lexicaux. Mais nous vous déconseillons fortement d'utiliser cette fonctionnalité. Son utilisation peut amener à des comportements inattendus et peut même faire planter certaines versions de Perl. Cette fonctionnalité est dépréciée et sera supprimée dans une version ultérieure.

Voir Paquetages in *perlmod* pour des plus amples informations à propos des packages (packages), des modules et des classes. Voir *perlsub* pour tous les problèmes de portée.

#### pipe READHANDLE,WRITEHANDLE

Ouvre une paire de tubes (pipe) exactement comme l'appel système correspondant. Remarquez qu'en faisant une boucle de tubes entre plusieurs processus vous risquez un inter-blocage (deadlock) à moins d'y faire très attention. De plus, sachez que les tubes de Perl utilisent des tampons d'entrée/sortie. Donc, selon les applications, vous aurez peut-être à positionner correctement `$|` pour vider vos `WRITEHANDLE` après chaque commande.

Voir *IPC::Open2*, *IPC::Open3* et Communication Bidirectionnelle avec un autre Processus in *perlipc* pour des exemples sur tout ça.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (close-on-exec) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de `$F`. Voir `$F` in *perlvar*.

#### pop TABLEAU

##### pop

Dépile et retourne la dernière valeur du tableau `TABLEAU`. La taille du tableau est diminué d'une unité. Cette commande a le même effet que :

```
$TABLEAU[$#TABLEAU--]
```

Si le tableau est vide, c'est la valeur `undef` qui est retournée (cela peut arriver dans d'autres cas aussi). Si `TABLEAU` est omis, `pop` dépilera soit le tableau `@ARGV` dans le programme principale soit `@_` dans les sous-routines, exactement comme `shift()`.

#### pos SCALAIRE

##### pos

Retourne l'offset (l'index du caractère dans la chaîne) où s'est arrêté la dernière recherche `m//g` sur la variable `SCALAIRE` spécifiée (s'appliquera à `$_` si la variable `SCALAIRE` n'est pas spécifiée). Notez que 0 est un offset valide. `unde` indique que la position de recherche a été réinitialisée (habituellement suite à une recherche infructueuse mais aussi, évidemment, si aucune recherche n'a encore été effectuée sur le scalaire). `pos` accède directement à l'emplacement où le moteur d'expression rationnelle stocke son offset si bien que l'affectation d'une valeur à `pos` modifiera cet offset et influencera donc l'assertion de longueur nulle `\G` d'une expression rationnelle. Comme l'échec d'un recherche par `m//gc` ne réinitialise pas l'offset, la valeur retournée par `pos` ne changera pas dans ce cas. Voir *perlre* et *perlop*.

#### print DESCRIPTEUR LISTE

##### print LISTE

**print**

Affiche une chaîne ou une liste de chaînes. Retourne true (vrai) en cas de succès. DESCRIPTEUR peut être une variable scalaire auquel cas cette variable contiendra le nom ou la référence du DESCRIPTEUR et donc introduira un niveau d'indirection supplémentaire. (REMARQUE : si DESCRIPTEUR est une variable et que l'élément syntaxique qui suit est un terme, il peut être interprété accidentellement comme un opérateur à moins d'ajouter un + ou de mettre des parenthèses autour des arguments.) Si aucun DESCRIPTEUR n'est spécifié, affichera par défaut sur la sortie standard (ou le dernier canal de sortie sélectionné – voir `select`). Si la liste LISTE est elle aussi omise, affichera \$\_ sur le canal de sortie courant. Pour utiliser un autre canal que STDOUT comme canal de sortie par défaut, utilisez l'opération `select`. La valeur courante de \$, (si elle existe) est affichée entre chaque item de LISTE. Remarquez qu'étant donné que `print` utilise une liste (LISTE), tout ce qui est dans LISTE est évalué dans un contexte de liste et, en particulier, toutes les expressions évaluées dans les sous-routines appelées le seront dans un contexte de liste. Faites aussi attention de ne pas faire suivre le mot-clé `print` par une parenthèse ouvrante à moins de vouloir clore la liste de ses arguments à la parenthèse fermante correspondante – sinon préfixez votre parenthèse par un + ou entourez tous les arguments par des parenthèses.

Notez que si vous stockez vos DESCRIPTEUR dans un tableau ou si vous utilisez une expression plus complexe qu'une simple variable scalaire pour le retrouver, vous devrez utiliser un bloc qui retourne la valeur :

```
print { $files[$i] } "stuff\n";
print { $OK ? STDOUT : STDERR } "stuff\n";
```

**printf DESCRIPTEUR FORMAT, LISTE****printf FORMAT, LISTE**

Équivalent à `print DESCRIPTEUR sprintf(FORMAT, LISTE)` sauf que \$\ (le séparateur d'enregistrements en sortie) n'est pas ajouté. Le premier argument de la liste sera interprété comme le format de `printf()`. Voir `sprintf` pour les explications concernant le format. Si `use locale` est actif, le caractère utilisé comme séparateur décimal pour les nombres réels sera dépendant de la valeur de locale spécifiée dans `LC_NUMERIC`. Voir *perllocale*.

Ne tombez pas de le piège d'utiliser `printf()` alors qu'un simple `print()` suffirait. `print()` est plus efficace et moins sujet à erreur.

**prototype FONCTION**

Retourne, sous forme de chaîne, le prototype d'une fonction (ou `undef` si la fonction n'a pas de prototype). FONCTION est une référence ou le nom de la fonction dont on veut retrouver le prototype.

Si FONCTION est une chaîne commençant par `CORE::`, la suite de la chaîne se réfère au nom d'une fonction interne de Perl. Si la fonction interne n'est pas *redéfinissable* (par exemple `qw//`) ou si ses arguments ne peuvent s'exprimer sous forme de prototype (par exemple `system()`) - en d'autres termes, si la fonction interne ne se comporte pas comme une fonction Perl - la fonction prototype retournera `undef`. Sinon, c'est la chaîne qui décrit le prototype qui est retournée.

**push TABLEAU,LISTE**

Traite TABLEAU comme une pile et empile les valeurs de la liste LISTE à la fin du tableau TABLEAU. La longueur de TABLEAU est augmentée de la longueur de la liste LISTE. Cela a le même effet que :

```
for $value (LISTE) {
 $TABLEAU[++$#TABLEAU] = $value;
}
```

mais en plus efficace. Retourne le nombre d'éléments présents dans le tableau une fois le push effectué.

**q/CHAINE/****qq/CHAINE/****qr/CHAINE/****qx/CHAINE/****qw/CHAINE/**

Guillemets/Apostrophes généralisées. Voir *perlop*.

**quotemeta EXPR****quotemeta**

Retourne la valeur de EXPR avec tous les caractères non alphanumériques précédés par un backslash (une barre oblique inverse). (Tous les caractères non reconnus par `/[A-Za-z_0-9]/` seront précédés d'un backslash quels que soient les réglages des locale). C'est la fonction interne qui implémente la séquence d'échappement `\Q` dans les chaînes entre guillemets.

Si EXPR est omis, s'appliquera à \$\_.

**rand EXPR****rand**

Retourne un nombre fractionnaire aléatoire plus grand ou égal à 0 et plus petit que la valeur de EXPR (la valeur de EXPR devrait être positive). À défaut de EXPR, c'est 1 qui est utilisé comme borne. Actuellement, si EXPR vaut 0, c'est aussi un cas spécial qui est équivalent à un appel avec 1 - ceci n'est documenté que depuis la version 5.8.0 de perl et peut changer dans des futures versions de perl. rand() appelle automatiquement srand() sauf si cela a déjà été fait. Voir aussi srand().

(Remarque : si votre fonction rand retourne régulièrement des nombres trop grands ou trop petits alors votre version de Perl a probablement été compilée avec une valeur erronée pour RANDBITS.)

**read DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET****read DESCRIPTEUR,SCALAIRE,LONGUEUR**

Essaye de lire LONGUEUR caractères depuis le DESCRIPTEUR spécifié et les stocke dans la variable SCALAIRE. Retourne le nombre de caractères réellement lus, 0 à la fin du fichier ou undef si une erreur a lieu (dans ce dernier cas, la cause de l'erreur est dans \$!). La taille de la variable SCALAIRE augmentera ou diminuera pour atteindre la taille exacte de ce qui est lu.

Un OFFSET (décalage) peut être spécifié pour placer les données lues ailleurs qu'au début de la chaîne. Un OFFSET négatif désignera un décalage compté à partir de la fin de la chaîne. Un OFFSET positif supérieur à la longueur de la chaîne complétera cette dernière avec des octets "\0" pour atteindre cette longueur avant d'y ajouter ce qui sera lu.

Cette fonction est implémentée par des appels à la fonction fread() de Perl ou du système. Pour obtenir un véritable appel système read(2), voir sysread().

Notez que l'on parle bien de caractères : selon l'état du DESCRIPTEUR, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les DESCRIPTEURs opèrent sur des octets mais, par exemple, si un DESCRIPTEUR a été ouvert avec le filtre d'entrée/sortie :utf8 (voir open et la directive open dans open) alors les entrées/sorties se feront sur des caractères Unicode codés en UTF-8 et non sur des octets. Ça marche de la même manière avec les directives :encoding.

**readdir DIRHANDLE**

Retourne l'entrée suivante d'un répertoire ouvert par opendir(). Dans un contexte de liste, retournera toutes les entrées restant dans le répertoire. Si il n'y a plus d'entrée, retournera la valeur undef dans un contexte scalaire ou une liste vide dans un contexte de liste.

Si vous prévoyez de faire des tests de fichiers sur les valeurs retournées par readdir(), n'oubliez pas de les préfixer par le répertoire en question. Sinon, puisqu'aucun appel à chdir() n'est effectué, vous risquez de tester un mauvais fichier.

```
opendir(DIR, $some_dir) || die "can't opendir $some_dir: $!";
@dots = grep { /\^\./ && -f "$some_dir/$_" } readdir(DIR);
closedir DIR;
```

**readline EXPR**

Lit à partir du descripteur dont le typeglob est donné par EXPR. Dans un contexte scalaire, chaque appel lit et retourne la ligne suivante jusqu'à atteindre la fin du fichier. À ce moment, un appel supplémentaire retournera undef. Dans un contexte de liste, la lecture se fera jusqu'à la fin du fichier et le résultat sera une liste de lignes. La notion de "ligne" utilisée ici est celle définie par la variable \$/ ou \$INPUT\_RECORD\_SEPARATOR. Voir \$/ in perlvar. Lorsque \$/ est positionné à undef, que readline() est utilisé dans un contexte scalaire (i.e. en mode 'slurp') et que le fichier lu est vide, le premier appel retourne "" et undef pour les suivants.

C'est la fonction interne qui implémente l'opérateur <EXPR> mais vous pouvez l'utiliser directement. L'opérateur <EXPR> est décrit en détail dans Les opérateurs d'E/S in perlop.

```
$line = <STDIN>;
$line = readline(*STDIN); # même chose
```

Si readline rencontre une erreur système, \$! contiendra le message d'erreur correspondant. Il peut être utile de vérifier \$! lorsque vous faites des lectures depuis un descripteur non sûr, comme un tty ou un socket. L'exemple suivant readline sous sa forme d'opérateur et contient les instructions permettant de vérifier que le readline s'est bien passé.

```
for (;;) {
 undef $!;
 unless (defined($line = <>)) {
 die $! if $!;
```

```

 last; # reached EOF
 }
 # ...
}

```

**readlink EXPR****readlink**

Retourne la valeur d'un lien symbolique si les liens symboliques sont implémentés. Sinon, produit une erreur fatale. Si il y a une erreur système, cette fonction retournera la valeur undef et positionnera la variable \$! (errno). Si EXPR est omis, s'applique à \$\_.

**readpipe EXPR**

EXPR est exécuté comme une commande système. La sortie standard de la commande est collectée puis retournée. Dans un contexte scalaire, cette sortie est une seule chaîne (contenant éventuellement plusieurs lignes). Dans un contexte de liste, retourne une liste de lignes (telles que définies par la variable \$/ ou \$INPUT\_RECORD\_SEPARATOR). C'est la fonction interne qui implémente l'opérateur qx/EXPR/ mais vous pouvez l'utiliser directement. L'opérateur qx/EXPR/ est décrit plus en détail dans Les opérateurs d'E/S in *perlop*.

**recv SOCKET,SCALAIRE,LONGUEUR,FLAGS**

Reçoit un message depuis un socket. Tente de recevoir LONGUEUR caractères de données dans la variable SCALAIRE depuis le descripteur spécifié par SOCKET. SCALAIRE grossira ou réduira jusqu'à la taille des données réellement lues. Utilise les mêmes FLAGS que l'appel système du même nom. Retourne l'adresse de l'émetteur si le protocole de SOCKET le permet ; retourne une chaîne vide sinon. Retourne la valeur undef en cas d'erreur. Cette fonction est implémentée en terme d'appel système recvfrom(2). Voir UDP : Transfert de Message in *perlipc* pour des exemples.

Notez que l'on parle bien de *caractères* : selon l'état du SOCKET, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les SOCKETS opèrent sur des octets mais si un SOCKET a été modifié via binmode() pour utiliser le filtre d'entrée/sortie :utf8 (voir la directive open et open) alors les entrées/sorties se feront sur des caractères Unicode codés en UTF-8 et non sur des octets. C'est similaire avec les directives :encoding.

**redo LABEL****redo**

La commande redo redémarre une boucle sans évaluer à nouveau la condition. Le bloc continue, s'il existe, n'est pas évalué. Si l'étiquette LABEL est omise, la commande se réfère à la boucle englobante la plus proche. Les programmes qui veulent réutiliser eux-mêmes ce qui vient juste d'être lu utilisent cette commande :

```

un programme simple pour supprimer les commentaires en Pascal
(Attention: suppose qu'il n'y pas de { ni de } dans les chaînes.)
LINE: while (<STDIN>) {
 while (s|({.*}.*){.*}|$1 |) {}
 s|{.*}| |;
 if (s|{.*}| |) {
 $front = $_;
 while (<STDIN>) {
 if (/}/) { # end of comment?
 s|^|$front\{|;
 redo LINE;
 }
 }
 }
}
print;
}

```

redo ne peut pas être utilisé pour redémarrer un bloc qui doit retourner une valeur comme eval {}, sub {} ou do {} et ne devrait pas être utilisé pour sortir d'une opération grep() ou map().

Remarques qu'un bloc en lui-même est sémantiquement identique à une boucle qui ne s'exécute qu'une fois. Donc redo dans un tel bloc le transforme effectivement en une construction de boucle.

Voir aussi continue pour illustrer la manière dont last, next et redo fonctionnent.

**ref EXPR****ref**

Retourne une chaîne non vide si EXPR est une référence et une chaîne vide sinon. Si EXPR n'est pas spécifié, s'applique à \$\_. La valeur retournée dépend du type de ce qui est référencé par la référence. Les types internes incluent :

```

SCALAR
ARRAY
HASH
CODE
REF
GLOB
LVALUE

```

Si l'objet référencé a été béni (par `bless()`) par un package alors le nom du package est retourné. Vous pouvez voir `ref()` comme une sorte d'opérateur `typeof()` (type de).

```

if (ref($r) eq "HASH") {
 print "r is a reference to a hash.\n";
}
unless (ref($r)) {
 print "r is not a reference at all.\n";
}

```

Voir aussi *perlref*.

### **rename ANCIENNOM,NOUVEAUNOM**

Change le nom d'un fichier ; un fichier préexistant de nom NOUVEAUNOM sera écrasé. Retourne true (vrai) en cas de succès ou false (faux) sinon.

Le comportement de cette fonction varie beaucoup d'un système à l'autre. Par exemple, habituellement elle ne fonctionne pas entre des systèmes de fichiers différents même si la commande système *mv* réussit parfois à le faire. Il y a d'autres restrictions selon que cela fonctionne ou non sur des répertoires, sur des fichiers ouverts ou sur des fichiers préexistants. Regarder *perlport* et la page de documentation de `rename(2)` pour les détails.

### **require VERSION**

### **require EXPR**

#### **require**

Exige une version de Perl spécifiée par VERSION ou une « sémantique » spécifiée par EXPR ou par \$\_ si EXPR n'est pas fourni.

VERSION peut être spécifié soit sous la forme d'une valeur numérique telle que 5.006, elle sera alors comparée à \$] soit sous la forme d'une valeur littérale telle que v5.6.1 qui sera alors comparée à \$^V (ou son synonyme \$PERL\_VERSION). Une erreur fatale est produite lors de l'exécution si VERSION est supérieure à la version de l'interpréteur Perl courant. Comparez avec `use` qui peut faire un contrôle similaire mais lors de la compilation.

```

require v5.6.1; # contrôle de version à l'exécution
require 5.6.1; # idem
require 5.006_001; # argument numérique autorisé par compatibilité

```

Sinon, `require` exige que le fichier d'une bibliothèque soit inclus si ce n'est pas déjà fait. Le fichier est inclus via le mécanisme `do-FICHER` qui est pratiquement une variante de `eval()`. Sa sémantique est similaire à la procédure suivante :

```

sub require {
 my ($filename) = @_;
 if (exists $INC{$filename}) {
 return 1 if $INC{$filename};
 die "Compilation failed in require";
 }
 ITER: {
 foreach $prefix (@INC) {
 $realfilename = "$prefix/$filename";
 if (-f $realfilename) {
 $INC{$filename} = $realfilename;
 $result = do $realfilename;
 last ITER;
 }
 }
 die "Can't find $filename in \@INC";
 }
 if ($@) {

```



```

 $INC{$filename} = undef;
 die $@;
} elsif (!$result) {
 delete $INC{$filename};
 die "$filename did not return true value";
} else {
 return $result;
}
}

```

Remarquez que le fichier ne sera pas inclus deux fois sous le même nom.

Le fichier doit retourner true (vrai) par sa dernière instruction pour indiquer une exécution correcte du code d'initialisation. Il est donc courant de terminer un tel fichier par un "1;" à moins d'être sûr qu'il retournera true (vrai) par un autre moyen. Mais il est plus sûr de mettre "1;" au cas où vous ajouteriez quelques instructions.

Si EXPR est un nom simple (bareword), require suppose l'extension ".pm" et remplace pour vous les "::" par des "/" dans le nom du fichier afin de rendre plus simple le chargement des modules standards. Cette forme de chargement des modules ne risque pas d'altérer votre espace de noms.

En d'autres termes, si vous dites :

```
require Foo::Bar; # un splendide mot simple
```

La fonction require cherchera en fait le fichier "Foo/Bar.pm" dans les répertoires spécifiés par le tableau @INC.

Mais si vous essayez :

```

$class = 'Foo::Bar';
require $class; # $class n'est pas un mot simple
#ou
require "Foo::Bar"; # n'est pas un mot simple à cause des guillemets

```

La fonction require cherchera le fichier "Foo::Bar" dans les répertoires du tableau @INC et se plaindra qu'elle ne peut trouver le fichier "Foo::Bar". Dans ce cas, vous pouvez faire :

```
eval "require $class";
```

Maintenant que vous savez comment require cherche les fichiers dans le cas d'un argument sous la forme d'un mot simple, il y a quelques fonctionnalités supplémentaires. Avant de chercher l'extension ".pm", require cherchera un fichier avec l'extension ".pmc". Un fichier avec une telle extension est supposé contenir du code binaire (du bytecode) obtenu par B::Bytecode. Si ce fichier est trouvé et si sa date de modification est plus récente que celle du fichier non-compilé ".pm" correspondant, il sera chargé à sa place.

Vous pouvez aussi accrocher des actions (on dit crochets ou "hooks") à la fonctionnalité d'importation en plaçant directement du code Perl dans le tableau @INC. Il y a trois formes de crochets : une référence vers une sous-routine, une référence vers un tableau et une référence vers un objet bénit.

La référence vers une sous-routine est le cas simple. Lorsque le système d'inclusion rencontre une telle référence en parcourant le tableau @INC, la sous-routine est appelée avec deux arguments. Le premier est une référence vers la sous-routine elle-même et le second est le nom du fichier à inclure (par exemple "Foo/Bar.pm"). La sous-routine devrait alors retourner soit undef soit un descripteur à partir duquel le fichier à inclure sera lu. Si la valeur de retour est undef, require continuera sa recherche dans la suite du tableau @INC>

Si le crochet est une référence vers un tableau, le premier élément de ce tableau doit être une référence vers une sous-routine. Cette sous-routine sera appelée comme ci-dessous mais le premier argument sera la référence vers le tableau. Cela permet de passer des arguments indirectement à la sous-routine.

En d'autres termes, vous pouvez écrire :

```

push @INC, \&my_sub;
sub my_sub {
 my ($coderef, $filename) = @_; # $coderef est \&my_sub
 ...
}

```

ou :

```

push @INC, [\&my_sub, $x, $y, ...];
sub my_sub {
 my ($arrayref, $filename) = @_;
 # On retrouve $x, $y, ...
}

```

```

 my @parameters = @$arrayref[1..$$arrayref];
 ...
}

```

Si c'est une référence vers un objet, cet objet doit avoir une méthode INC qui sera appelée comme ci-dessus avec comme premier paramètre l'objet lui-même. (Notez que vous devez complètement qualifier le nom de cette subroutine pour éviter qu'il soit placé de force dans le package main.) Voici un exemple typique de code :

```

Dans Foo.pm
package Foo;
sub new { ... }
sub Foo::INC {
 my ($self, $filename) = @_;
 ...
}

Dans le programme principal (main)
push @INC, new Foo(...);

```

Notez que ces crochets ont aussi le droit de remplir les données dans %INC correspondant aux fichiers qu'ils ont chargés. Voir %INC in *perlvar*.

Pour une fonctionnalité d'importation encore plus puissante, voir *use* et *perlmod*.

### reset EXPR

#### reset

Généralement utilisée dans un bloc *continue* à la fin d'une boucle pour effacer les variables et réinitialiser les recherches ?? pour qu'elle marche à nouveau. L'expression EXPR est interprétée comme une liste de caractères (le moins est autorisé pour des intervalles). Toutes les variables commençant par l'un de ces caractères sont réinitialisées à leur état primitif. Si EXPR est omis, les motifs de recherche qui ne marchent qu'une fois (?motif?) sont réinitialisés pour fonctionner à nouveau. Ne réinitialise que les variables et les motifs du package courant. Retourne toujours 1. Exemples :

```

reset 'X'; # réinitialise toutes les variables X...
reset 'a-z'; # réinitialise toutes les variables
 # commençant par une minuscule
reset; # réinitialise juste les motifs ?...?

```

Réinitialiser "A-Z" n'est pas recommandé parce que cela efface les tableaux @ARGV et @INC ainsi que la table de hachage %ENV. Ne réinitialise que les variables de package – les variables lexicales ne sont pas modifiées mais elles s'effacent toutes seules dès que l'on sort de leur portée, ce qui devrait vous inciter à les utiliser. Voir *my*.

### return EXPR

#### return

Sort d'une subroutine, d'un bloc *eval()* ou d'un *do FICHIER* en retournant la valeur donnée par EXPR. L'évaluation de EXPR peut se faire dans un contexte scalaire, de liste ou vide selon la manière dont la valeur sera utilisée. Le contexte peut varier d'une exécution à l'autre (voir *wantarray()*). Si aucune EXPR n'est donnée, retourne la liste vide dans un contexte de liste, la valeur *undef* dans un contexte scalaire et rien du tout dans un contexte vide. (Remarque : en l'absence de *return* explicite, une subroutine, un bloc *eval* ou un *do FICHIER* retournera automatiquement la valeur de la dernière expression évaluée.)

### reverse LISTE

Dans un contexte de liste, retourne une liste de valeurs constituée des éléments de LISTE en ordre inverse. Dans un contexte scalaire, concatène les éléments de LISTE et retourne la chaîne ainsi constituée mais avec les caractères dans l'ordre inverse.

```

print reverse <>; # tac (cat à l'envers) les lignes,
 # la dernière ligne en premier

undef $/; # pour un <> efficace
print scalar reverse <>; # tac (cat à l'envers) les octets,
 # la dernière ligne en reimerp

```

Utilisée sans argument dans un contexte scalaire, la fonction *reverse* inverse \$\_.

Cet opérateur est aussi utilisé pour inverser des tables de hachage bien que cela pose quelques problèmes. Si une valeur est dupliquée dans la table originale, seule l'une des ces valeurs sera représentée comme une clé dans la table résultante. Cela nécessite aussi de mettre toute la table à plat avant d'en reconstruire une nouvelle ce qui peut prendre beaucoup de temps sur une grosse table telle qu'un fichier DBM.

```
%by_name = reverse %by_address; # Inverse la table
```

**rewinddir DIRHANDLE**

Ramène la position courante au début du répertoire pour le prochain `readdir()` sur `DIRHANDLE`.

**rindex CHAINE,SUBSTR,POSITION****rindex CHAINE,SUBSTR**

Fonctionne exactement comme `index` sauf qu'il retourne la position de la *dernière* occurrence de `SUBSTR` dans `CHAINE`. Si `POSITION` est spécifiée, retourne la dernière occurrence commençant avant ou exactement à cette position.

**rmdir REPNOM****rmdir**

Efface le répertoire spécifié par `REPNOM` si ce répertoire est vide. En cas de succès, retourne `true` (vrai) ou sinon, retourne `false` (faux) et positionne la variable `!` (`errno`). Si `REPNOM` est omis, utilise `$_`.

**s///**

L'opérateur de substitution. Voir *perlop*.

**scalar EXPR**

Contraint l'interprétation de `EXPR` dans un contexte scalaire et retourne la valeur de `EXPR`.

```
@counts = (scalar @a, scalar @b, scalar @c);
```

Il n'y pas d'opérateur équivalent pour contraindre l'interprétation d'une expression dans un contexte de liste parce qu'en pratique ce n'est jamais nécessaire. Si vous en avez réellement besoin, vous pouvez utiliser une construction comme `@{ (une expression) }` mais un simple `(une expression)` suffit en général.

Comme `scalar` est un opérateur unaire, si vous l'utilisez accidentellement sur une `EXPR` constituée d'une liste entre parenthèses, cela se comporte comme l'opérateur scalaire virgule en évaluant tous les éléments dans un contexte vide sauf le dernier élément qui est évalué dans un contexte scalaire et retourné. C'est rarement ce que vous vouliez.

L'instruction suivante :

```
print uc(scalar(&foo,$bar)), $baz;
```

est équivalente à ces deux instructions :

```
&foo;
print(uc($bar), $baz);
```

Voir *perlop* pour les détails sur les opérateurs unaires et l'opérateur virgule.

**seek DESCRIPTEUR,POSITION,WHENCE**

Modifie la position d'un `DESCRIPTEUR` exactement comme le fait l'appel `fseek()` de `stdio()`. `DESCRIPTEUR` peut être une expression dont la valeur donne le nom du descripteur. Les valeurs possibles de `WHENCE` sont `0` pour régler la nouvelle position *en octets* à `POSITION`, `1` pour la régler à la position courante plus `POSITION` ou `2` pour la régler à EOF plus `POSITION` (en général négative). Pour `WHENCE`, vous pouvez utiliser les constantes `SEEK_SET`, `SEEK_CUR` et `SEEK_END` (relatif au début du fichier, à la position courante, à la fin du fichier) provenant du module `Fcntl`. Renvoie `1` en cas de succès et `0` sinon.

Notez bien qu'on parle *en octets* : même si le `DESCRIPTEUR` a été configuré pour opérer sur des caractères (par exemple en utilisant le filtre `:utf8`), `seek()` travaille quand même en terme d'octets et non en terme de caractères (implémenter cette fonctionnalité aurait rendu `seek()` et `tell()` extrêmement lents).

Si vous voulez régler la position pour un fichier dans le but d'utiliser `sysread()` ou `syswrite()`, n'utilisez pas `seek()` – la bufferisation rend ses effets imprévisibles et non portables. Utilisez `sysseek()` à la place.

À cause des règles et de la rigueur du C ANSI, sur certains systèmes, vous devez faire un `seek` à chaque fois que vous basculez entre lecture et écriture. Entre autres choses, cela a pour effet d'appeler la fonction `clearerr(3)` de `stdio`. Un `WHENCE` de `1` (`SEEK_CUR`) est pratique pour ne pas modifier la position dans le fichier :

```
seek(TEST, 0, 1);
```

C'est aussi très pratique pour les applications qui veulent simuler `tail -f`. Une fois rencontré un EOF en lecture et après avoir attendu un petit peu, vous devez utiliser `seek()` pour réactiver les choses. L'appel à `seek()` ne modifie pas la position courante mais par contre, il efface la condition fin-de-fichier (EOF) sur le descripteur et donc, au prochain `<FILE>`, Perl essaiera à nouveau de lire quelque chose. Espérons-le.

Si cela ne marche pas (certaines implémentations des E/S sont particulièrement irascibles) alors vous devrez faire quelque chose comme :

```

for (;;) {
 for ($curpos = tell(FILE); $_ = <FILE>;
 $curpos = tell(FILE)) {
 # search for some stuff and put it into files
 }
 sleep($for_a_while);
 seek(FILE, $curpos, 0);
}

```

**seekdir DIRHANDLE,POS**

Règle la position courante pour la routine `readdir()` sur un `DIRHANDLE`. `POS` doit être une valeur retournée par `telldir()`. `seekdir` possède les mêmes limitations que l'appel système correspondant.

**select DESCRIPTEUR****select**

Retourne le descripteur courant. Sélectionne le descripteur `DESCRIPTEUR` comme sortie par défaut si `DESCRIPTEUR` est fourni. Ceci a deux effets : tout d'abord, un `write()` ou un `print()` sans descripteur spécifié iront par défaut sur ce `DESCRIPTEUR`. Ensuite, toutes références à des variables relatives aux sorties se référeront à ce canal de sortie. Par exemple, si vous devez spécifier un en-tête de format pour plusieurs canaux de sortie, vous devez faire la chose suivante :

```

select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';

```

`DESCRIPTEUR` peut être une expression dont le résultat donne le nom du descripteur réel. Donc :

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Certains programmeurs préfèrent considérer les descripteurs comme des objets avec des méthodes. Ils écriraient donc l'exemple précédent de la manière suivante :

```

use IO::Handle;
STDERR->autoflush(1);

```

**select RBITS,WBITS,EBITS,TIMEOUT**

Ceci utilise directement l'appel système `select(2)` avec les masques de bit spécifiés qui peuvent être construits en utilisant `fileno()` et `vec()` comme dans les lignes suivantes :

```

$rin = $win = $ein = '';
vec($rin, fileno(STDIN), 1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;

```

Si vous voulez surveiller de nombreux descripteurs, vous aurez peut-être à écrire une sous-routine :

```

sub fhbits {
 my(@fhlist) = split(' ', $_[0]);
 my($bits);
 for (@fhlist) {
 vec($bits, fileno($_), 1) = 1;
 }
 $bits;
}
$rin = fhbits('STDIN TTY SOCK');

```

L'appel classique est :

```

($nfound, $timeleft) =
 select($rout=$rin, $wout=$win, $eout=$ein, $timeout);

```

ou pour attendre que quelque chose soit prêt :

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

De nombreux systèmes ne prennent pas la peine de retourner quelque chose d'utile dans `$timeleft` (le temps restant). En conséquence, un appel à `select()` dans un contexte scalaire retourne juste `$nfound`.

`undef` est une valeur acceptable pour les masques de bits. Le timeout, s'il est spécifié, est donné en secondes et peut être fractionnaire. Note : certaines implémentations ne sont pas capables de retourner `$timeleft`. Dans ce cas, elles retournent toujours un `$timeleft` égal au `$timeout` fourni.

Vous pouvez spécifier une attente de 250 millisecondes de la manière suivante :

```
select(undef, undef, undef, 0.25);
```

Notez que selon les implémentations, un `select` reprendra ou non après réception d'un signal (un SIGALRM par exemple). Voir aussi *perlport* pour des informations concernant la portabilité de `select`.

En cas d'erreur, `select` se comporte comme l'appel système `select(2)` : il retourne -1 et définit `!`.

Note : sur certains Unix, l'appel système `select(2)` peut indiquer qu'un descripteur de socket est "prêt pour la lecture" alors qu'il n'y a en fait rien à lire, rendant ainsi la prochaine lecture bloquante. Cela peut être évité en utilisant systématiquement l'option `O_NONBLOCK` sur ce socket. Voir `select(2)` et `fcntl(2)` pour plus de détails.

**ATTENTION** : il ne faut pas mélanger des E/S bufferisées (comme `read()` ou `<FH>`) avec `select()` excepté lorsque la norme POSIX le permet et, dans ce cas, uniquement sur les systèmes POSIX. Vous devez utiliser `sysread()` à la place.

### **semctl ID,SEMNUM,CMD,ARG**

Appelle la fonction `semctl()` des IPC System V. Vous aurez sans doute besoin de :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si `CMD` est `IPC_STAT` ou `GETALL` alors `ARG` doit être une variable capable de contenir la structure `semid_ds` retournée ou le tableau des valeurs des sémaphores. Les valeurs retournées sont comme celles de `ioctl()` : la valeur `undef` en cas d'erreur, la chaîne `"0 but true"` pour rien ou la vraie valeur retournée dans les autres cas. `ARG` doit être un vecteur d'entiers courts (short) natifs qui peut être créé par `pack("s!", (0)x$nsem)`. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

### **semget KEY,NSEMS,FLAGS**

Appelle la fonction `semget()` des IPC System V. Renvoie l'id du sémaphore ou la valeur `undef` en cas d'erreur. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

### **semop KEY,OPSTRING**

Appelle la fonction `semop()` des IPC System V pour réaliser certaines opérations sur les sémaphores comme l'attente ou la signalisation. `OPSTRING` doit être un tableau compacté (par `pack()`) de structures `semop`. Chaque structure `semop` peut être générée par `pack("s!3", $semnum, $semop, $semflag)`. La taille de `OPSTRING` détermine le nombre total d'opérations sur les sémaphores. Renvoie `true` (vrai) en cas de succès ou `false` (faux) en cas d'erreur. Par exemple, le code suivant attend sur le sémaphore `$semnum` de l'ensemble de sémaphores d'id `$semid` :

```
$semop = pack("s!3", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

Pour envoyer un signal au sémaphore, remplacez le -1 par 1. Voir aussi la documentation de `IPC::SysV` et de `IPC::SysV::Semaphore`.

### **send SOCKET,MSG,FLAGS,TO**

#### **send SOCKET,MSG,FLAGS**

Envoie un message sur un socket. Envoie le scalaire `MSG` vers le descripteur `SOCKET`. Utilise les mêmes flags que l'appel système du même nom. Pour des sockets non connectés, vous devez spécifier dans le paramètre `TO` une destination pour l'envoi. Dans ce cas c'est la fonction C `sendto()` qui est utilisée. Retourne le nombre de caractères envoyés ou la valeur `undef` s'il y a une erreur. L'appel système `sendmsg(2)` n'est pas implémenté pour l'instant. Voir les exemples dans `UDP : Transfert de Message in perlipc`.

Notez que l'on parle bien de *caractères* : selon l'état du `SOCKET`, ce seront soit des octets (8-bit) soit des caractères qui seront lus. Par défaut tous les `SOCKETs` opèrent sur des octets mais si un `SOCKET` a été modifié via `binmode()` pour utiliser le filtre d'entrée/sortie `:utf8` (voir `open`, la directive `open` et `open`) alors les entrées/sorties se feront sur des caractères Unicode codés en UTF-8 et non sur des octets. La directive `:encoding` a des effets similaires.

### **setpgrp PID,PGRP**

Modifie le groupe de processus courant pour le `PID` spécifié (0 pour le processus courant). Cela produira une erreur fatale si vous l'utilisez sur un système qui n'implémente pas l'appel POSIX `setpgid(2)` ou l'appel BSD `setpgrp(2)`. Si les arguments sont omis, ils prennent comme valeur par défaut 0,0. Remarquez aussi que la version BSD 4.2 de `setpgrp()` n'accepte aucun argument. Donc seul `setpgrp 0,0` est portable. Voir aussi `POSIX::setsid()`.

**setpriority WHICH,WHO,PRIORITY**

Modifie la priorité courante d'un process, d'un groupe de process ou d'un utilisateur. (Voir `setpriority(2)`.) Cela produira une erreur fatale si vous l'utilisez sur un système qui n'implémente pas `setpriority(2)`.

**setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL**

Modifie l'option spécifiée d'un socket. Retourne undef en cas d'erreur. Utilise les constantes fournies par le module Socket pour LEVEL et OPTNAME. Les valeurs pour LEVEL peuvent aussi être récupérées via `getprotobyname`. OPTVAL peut être un chaîne compacté (par `pack()`) ou une valeur entière. Une valeur entière pour OPTVAL est un raccourci pour `pack("i", OPTVAL)`.

Voici un exemple qui désactive l'algorithme de Nagle pour un socket :

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

**shift TABLEAU****shift**

Retourne la première valeur d'un tableau après l'avoir supprimée du tableau en rétrécissant sa taille de 1 et en déplaçant tout vers le bas. Renvoie undef si il n'y a pas d'éléments dans le tableau. Si TABLEAU est omis, shift agira soit sur le tableau @\_ s'il est dans la portée lexicale d'une subroutine ou d'un format, soit sur le tableau @ARGV s'il est dans la portée lexicale d'un fichier ou s'il est dans une portée lexicale établie par l'une des constructions `eval "`, `BEGIN {}`, `END {}` ou `INIT {}`.

Voir aussi `unshift`, `push` et `pop`. `shift()` et `unshift()` agissent sur le côté gauche d'un tableau exactement comme `pop()` et `push()` le font sur le côté droit.

**shmctl ID,CMD,ARG**

Appelle la fonction `shmctl` de IPC System V. Vous aurez sans doute besoin de :

```
use IPC::SysV;
```

au préalable pour avoir les définitions correctes des constantes. Si CMD est `IPC_STAT` alors ARG doit être une variable capable de contenir la structure `shmid_ds` retournée. Les valeurs retournées sont comme celles de `ioctl()` : la valeur undef en cas d'erreur, la chaîne "0 but true" pour zéro ou la vraie valeur retournée dans les autres cas. Voir aussi la documentation de `IPC::SysV`.

**shmget KEY,SIZE,FLAGS**

Appelle la fonction `shmget` de IPC System V. Renvoie l'id du segment de mémoire partagée ou undef en cas d'erreur. Voir aussi la documentation de `IPC::SysV`.

**shmread ID,VAR,POS,SIZE****shmwrite ID,CHAINE,POS,SIZE**

Lit ou écrit le segment de mémoire partagée System V d'id ID en commençant à la position POS et sur une taille de SIZE en s'attachant à lui puis en le lisant ou en l'écrivant puis enfin en s'en détachant. Lors d'une lecture, VAR doit être une variable qui contiendra les données lues. Lors d'une écriture, si CHAINE est trop long, seuls SIZE octets seront utilisés, si CHAINE est trop court des caractères nuls seront ajoutés pour compléter jusqu'à SIZE octets. Retourne true (vrai) en cas de succès et false (faux) en cas d'erreur. `shmread()` souille (taint) la variable CHAINE. Voir aussi la documentation de `IPC::SysV`.

**shutdown SOCKET,HOW**

Ferme une connexion socket de la manière indiquée par HOW qui est interprété comme le fait l'appel système du même nom.

```
shutdown(SOCKET, 0); # J'ai arrêté de lire des données
shutdown(SOCKET, 1); # J'ai arrêté d'écrire des données
shutdown(SOCKET, 2); # J'ai arrêté d'utiliser ce socket
```

C'est pratique pour des sockets pour lesquels vous voulez indiquer à l'autre extrémité que vous avez fini d'écrire mais pas de lire ou vice versa. C'est aussi une forme plus insistante de `close` puisque qu'elle désactive aussi le descripteur de fichier pour tous les process dupliqués par `fork`.

**sin EXPR****sin**

Retourne le sinus de EXPR (exprimé en radians). Si EXPR est omis, retourne le sinus de \$\_.

Pour calculer la fonction inverse du sinus, vous pouvez utiliser la fonction `Math::Trig::asin` ou utiliser cette relation :

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

**sleep** **EXPR****sleep**

Demande au script de s'endormir pendant EXPR secondes ou pour toujours si EXPR est omis. Peut être interrompu si le process reçoit un signal comme SIGALRM. Renvoie la durée réelle du sommeil en secondes. Vous ne pourrez probablement pas mélanger des appels alarm() et sleep() car sleep() est souvent implémenté en utilisant alarm().

Sur quelques vieux systèmes, il se peut que la durée du sommeil soit d'une seconde de moins que celle que vous avez demandée en fonction de la manière dont il compte les secondes. Les systèmes plus modernes s'endorment toujours pendant la bonne durée. En revanche, il peut arriver que votre sommeil dure plus longtemps que prévu sur un système multi-tâches très chargé.

Pour des délais d'une granularité inférieure à la seconde, vous pouvez utiliser l'interface Perl syscall() pour accéder à setitimer(2) si votre système le supporte ou sinon regarder select() plus haut. Le module Time::HiRes (disponible sur CPAN ou intégré à la distribution standard depuis Perl 5.8.0) peut aussi aider.

Regarder aussi la fonction sigpause() du module POSIX.

**socket** **SOCKET,DOMAIN,TYPE,PROTOCOL**

Ouvre un socket du type spécifié et l'attache au descripteur SOCKET. DOMAIN, TYPE et PROTOCOL sont spécifiés comme pour l'appel système du même nom. Vous devriez mettre "use Socket;" au préalable pour importer les définitions correctes. Voir les exemples dans Sockets : Communication Client/Serveur in *perlipc*.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (close-on-exec) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de \$^F. Voir \$^F in *perlvar*.

**socketpair** **SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL**

Crée une paire de sockets sans nom dans le domaine spécifié et du type spécifié. DOMAIN, TYPE et PROTOCOL sont spécifiés comme pour l'appel système du même nom. Si l'appel système n'est pas implémenté, cela produit une erreur fatale. Retourne true (vrai) en cas de succès.

Sur les systèmes qui supportent le drapeau fermeture-à-l-exécution (close-on-exec) sur les fichiers, ce drapeau sera positionné pour de nouveaux descripteurs de fichier en fonction de la valeur de \$^F. Voir \$^F in *perlvar*.

Certains systèmes définissent pipe() en terme de socketpair(), auquel cas un appel à pipe(Rdr, Wtr) est quasiment équivalent à :

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1); # plus d'écriture pour le lecteur
shutdown(Wtr, 0); # plus de lecture pour l'écrivain
```

Voir *perlipc* pour des exemples d'utilisation de socketpair. Perl 5.8 et plus émuleront socketpair en utilisant de sockets IP sur la machine locale si votre système implémente les sockets mais pas socketpair.

**sort** **SUBNAME LISTE****sort** **BLOC LISTE****sort** **LISTE**

Dans un contexte de liste, sort() trie LISTE et retourne la liste de valeurs triée. Dans un contexte scalaire, le comportement de sort() est indéfini.

Si SUBNAME et BLOC sont omis, le tri est effectué dans l'ordre standard de comparaison de chaînes. Si SUBNAME est spécifié, il donne le nom d'une subroutine qui retourne un entier plus petit, égal ou plus grand que 0 selon l'ordre dans lequel les éléments du tableau doivent être triés. (Les opérateurs <=> et cmp sont extrêmement utiles dans de telles subroutines.) SUBNAME peut être une variable scalaire, auquel cas sa valeur donne le nom de (ou la référence vers) la subroutine à utiliser. À la place de SUBNAME, vous pouvez fournir un BLOC comme subroutine de tri anonyme et en ligne.

Si le prototype de la subroutine est (\$\$), les éléments à comparer sont passés par référence dans @\_ comme pour une subroutine normale. C'est plus lent qu'une subroutine sans prototype pour laquelle les éléments à comparer sont passés à la subroutine par les variables globales du package courant \$a et \$b (voir exemples ci-dessous). Dans ce dernier cas, il est contre-productif de déclarer \$a et \$b comme des variables lexicales.

Dans tous les cas, la subroutine ne peut pas être récursive. Les valeurs à comparer sont toujours passées par référence et ne devraient pas être modifiées.

Vous ne pouvez pas non plus sortir du bloc sort ou de la subroutine en utilisant un goto() ou les opérateurs de contrôle de boucles décrits dans *perlsyn*.

Lorsque use locale est actif, sort LISTE trie LISTE selon l'ordre (collation) du locale courant. Voir *perllocale*. La fonction sort retourne des alias de la liste originale, exactement comme la variable d'une boucle for qui est un alias de la liste d'éléments. Cela implique que la modification d'un élément d'une liste retournée par sort() (par

exemple via un `foreach`, un `map` ou un `grep`) modifiera réellement l'élément de la liste originale. Habituellement, c'est quelque chose à éviter si on veut écrire du code propre.

Jusqu'à Perl 5.6, Perl utilisait un algorithme quicksort pour implémenter le tri. Cet algorithme n'était pas stable et *pouvait* parfois être quadratique. (Un tri *stable* préserve l'ordre des éléments égaux. De plus, bien que la complexité moyenne d'un quicksort soit en  $O(N \log N)$  pour un tableau de taille  $N$ , la complexité peut atteindre  $O(N^2)$ , un comportement quadratique, pour certains tableaux.) En Perl 5.7, l'implémentation du quicksort a été remplacée par un algorithme stable mergesort dont la complexité est  $O(N \log N)$ . Mais des tests de performance ont montré que, dans certains cas et sur certaines plateformes, quicksort était plus rapide. Perl 5.8 propose donc une directive `sort` afin de choisir l'implémentation. Mais cela n'est pas satisfaisant et disparaîtra certainement dans une version future au profit d'un moyen permettant de caractériser le type d'entrée ou de sortie indépendamment de l'implémentation. Voir `use`.

Exemples :

```
tri alphabétique
@articles = sort @files;

idem mais avec une routine de tri explicite
@articles = sort {$a cmp $b} @files;

idem mais indépendant de la casse
@articles = sort {uc($a) cmp uc($b)} @files;

idem mais dans l'ordre inverse
@articles = sort {$b cmp $a} @files;

tri numérique ascendant
@articles = sort {$a <=> $b} @files;

tri numérique descendant
@articles = sort {$b <=> $a} @files;

tri de %age par valeur plutôt que par clé
en utilisant une fonction en-ligne
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;

tri utilisant le nom explicite d'une subroutine
sub byage {
 $age{$a} <=> $age{$b}; # supposé numérique
}
@sortedclass = sort byage @class;

sub backwards { $b cmp $a; }
@harry = qw(dog cat x Cain Abel);
@george = qw(gone chased yz Punished Axed);
print sort @harry;
 # affiche AbelCaincatdogx
print sort backwards @harry;
 # affiche xdogcatCainAbel
print sort @george, 'to', @harry;
 # affiche AbelAxedCainPunishedcatchaseddoggonetoxyz

tri inefficace par ordre numérique descendant utilisant
le premier entier après le signe = ou l'ensemble de
l'enregistrement si cet entier n'existe pas
@new = sort {
 ($b =~ /\=(\d+)/)[0] <=> ($a =~ /\=(\d+)/)[0]
 ||
 uc($a) cmp uc($b)
} @old;

la même chose mais plus efficace;
nous construisons un tableau auxiliaire d'indices
pour aller plus vite
@nums = @caps = ();
for (@old) {
```



```

 push @nums, /=(\d+)/;
 push @caps, uc($_);
}

@new = @old[sort {
 $nums[$b] <=> $nums[$a]
 ||
 $caps[$a] cmp $caps[$b]
} 0..$#old
];

même chose sans utiliser de variables temporaires
@new = map { $_->[0] }
 sort { $b->[1] <=> $a->[1]
 ||
 $a->[2] cmp $b->[2]
 } map { [$_, /=(\d+)/, uc($_)] } @old;

l'utilisation d'un prototype vous permet d'utiliser
n'importe quelle subroutine de comparaison comme
subroutine de tri (y compris des sous-routines d'autres packages)
package other;
sub backwards ($$) { $_[1] cmp $_[0]; } # $a and $b are not set here

package main;
@new = sort other::backwards @old;

tri stable, sans choix de l'algorithme
use sort 'stable';
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;

utilisation forcée d'un mergesort (non portable en dehors de Perl 5.8)
use sort '_mergesort'; # notez le _ décourageant
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;

```

Si vous utilisez `strict`, vous *NE DEVEZ PAS* déclarer `$a` et `$b` comme variables lexicales. Ce sont des variables globales au package. Cela signifie que si vous êtes dans le package `main`, c'est :

```
@articles = sort {$main::b <=> $main::a} @files;
```

ou juste :

```
@articles = sort {$::b <=> $::a} @files;
```

mais si vous êtes dans le package `FooPack`, c'est <:>

```
@articles = sort {$FooPack::b <=> $FooPack::a} @files;
```

La fonction de comparaison doit se comporter correctement. Si elle retourne des résultats incohérents (parfois elle dit que `$x[1]` est plus petit que `$x[2]` et d'autres fois le contraire par exemple), le résultat du tri n'est pas bien défini.

Puisque `<=>` retourne `undef` lorsque l'un de ses opérandes est `NaN` (not-a-number, pas un nombre) et puisque `sort` provoque une erreur fatale si le résultat d'une comparaison n'est pas défini, lorsque vous triez effectuez un tri en utilisant une fonction de comparaison comme `$a <=> $b`, faites attention aux listes contenant un `NaN`. L'exemple suivant exploite le fait que `NaN != NaN` pour éliminer les `NaN` de `@input`.

```
@result = sort { $a <=> $b } grep { $_ == $_ } @input;
```

### **splice TABLEAU,OFFSET,LONGUEUR,LISTE**

#### **splice TABLEAU,OFFSET,LONGUEUR**

#### **splice TABLEAU,OFFSET**

#### **splice TABLEAU**

Supprime d'un tableau les éléments consécutifs désignés par `OFFSET` (indice du premier élément) et `LONGUEUR` (nombre d'éléments concernés) et les remplace par les éléments de `LISTE` si il y en a. Dans un contexte de liste, renvoie les éléments supprimés du tableau. Dans un contexte scalaire, renvoie le dernier élément supprimé ou `undef` si aucun élément n'est supprimé. Le tableau grossit ou diminue si nécessaire. Si `OFFSET` est négatif, il est compté à

partir de la fin du tableau. Si LONGUEUR est omis, supprime tout à partir de OFFSET. Si LONGUEUR est négatif, supprime les éléments à partir de OFFSET en laissant -LONGUEUR éléments à la fin du tableau. Si OFFSET et LONGUEUR sont omis, supprime tout ce qui est dans le tableau. Si OFFSET va au-delà de la fin du tableau, perl produit un message d'avertissement et splice agit à la fin du tableau.

Les équivalences suivantes sont vraies en supposant que `<<[$ == 0 and $#a = $i] :>>`

```
push(@a, $x, $y) splice(@a, @a, 0, $x, $y)
pop(@a) splice(@a, -1)
shift(@a) splice(@a, 0, 1)
unshift(@a, $x, $y) splice(@a, 0, 0, $x, $y)
$a[$i] = $y splice(@a, $i, 1, $y)
```

Exemple, en supposant que la longueur des tableaux est passée avant chaque tableau :

```
sub aeq { # compare deux listes de valeurs
 my(@a) = splice(@_, 0, shift);
 my(@b) = splice(@_, 0, shift);
 return 0 unless @a == @b; # même longueur ?
 while (@a) {
 return 0 if pop(@a) ne pop(@b);
 }
 return 1;
}
if (&aeq($len, @foo[1..$len], 0+@bar, @bar)) { ... }
```

### **split /MOTIF/EXPR,LIMITE**

### **split /MOTIF/EXPR**

### **split /MOTIF/**

### **split**

Découpe la chaîne EXPR en une liste de chaînes et la retourne. Par défaut, les champs vides du début sont conservés et ceux de la fin sont éliminés (si tous les champs sont vides, ils sont considérés comme étant à la fin).

Si l'appel n'est pas dans un contexte de liste, split retourne le nombre de champs trouvés et les place dans le tableau @\_. (Dans un contexte de liste, vous pouvez forcer l'utilisation du tableau @\_ en utilisant ?? comme motif délimiteur mais il renvoie encore la liste des valeurs.) En revanche, l'utilisation implicite de @\_ par split est désapprouvée parce que cela écrase les arguments de votre sous-routine.

SI EXPR est omis, split découpe la chaîne \$\_. Si MOTIF est aussi omis, il découpe selon les blancs (après avoir sauté d'éventuels blancs au départ). Tout ce qui reconnu par MOTIF est considéré comme étant un délimiteur de champs. (Remarquez que le délimiteur peut être plus long qu'un seul caractère.)

Si LIMITE est spécifié et positif, fixe le nombre maximum de champs du découpage (le nombre de champs dépend du nombre de fois où le MOTIF sera reconnu dans EXPR). Si LIMITE n'est pas spécifié ou vaut zéro, les champs vides de la fin sont supprimés (choses dont les utilisateurs potentiels de pop() devraient se souvenir). Si LIMITE est négatif, il est traité comme si LIMITE avait une valeur arbitrairement très grande. Notez que découper une EXPR dont la valeur est la chaîne vide retourne toujours une liste vide, que LIMITE soit spécifié ou non.

Un motif qui peut correspondre à la chaîne vide (ne pas confondre avec le motif vide // qui n'est qu'un motif parmi tous ceux qui peuvent correspondre à la chaîne vide) découpera la valeur de EXPR en caractères séparés à chaque point où il sera reconnu. Par exemple :

```
print join(':', split(/ */, 'hi there'));
```

produira la sortie `hi:there`.

Il y a un cas spécial pour split, c'est l'utilisation du motif vide // qui ne reconnaît que la chaîne nulle et qui ne doit pas être confondue avec l'utilisation normale de // pour signifier "le dernier motif ayant été reconnu". Donc, pour split, l'exemple suivant :

```
print join(':', split(/, 'hi there'));
```

produira la sortie `hi: :there`.

Un champ vide de début (ou de fin) n'est produit que lorsqu'il y a reconnaissance du MOTIF avec une longueur positive au début (à la fin) de la chaîne. Une reconnaissance de longueur nulle en début ou en fin de chaîne ne produira pas de champs vides. Par exemple :

```
print join(':', split/(?=\w)/, 'hi there!'));
```

produira la sortie 'hi:t:h:e:r:e!'.

Le paramètre LIMITE peut être utilisé pour découper partiellement une ligne :

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

Lors de l'affectation à une liste, si LIMITE est omis ou nulle, Perl agit comme si LIMITE était juste supérieure au nombre de variables de la liste pour éviter tout travail inutile. Pour la liste ci-dessous, la valeur par défaut de LIMITE serait 4. Dans les applications où le temps est critique, il vous incombe de ne pas découper en plus de champs que ceux réellement nécessaires.

Si MOTIF contient des parenthèses (et donc des sous-motifs), un élément supplémentaire est créé dans le tableau résultat pour chaque chaîne reconnue par le sous-motif.

```
split(/([\s-]+)/, "1-10,20", 3);
```

produit la liste de valeurs

```
(1, '-', 10, ',', 20)
```

Si vous avez dans la variable \$header tout l'en-tête d'un email normal d'UNIX, vous devriez pouvoir le découper en champs et valeurs en procédant comme suit :

```
$header =~ s/\n\s+/ /g; # fix continuation lines
%hdrs = (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

Le motif /MOTIF/ peut être remplacé par une expression pour spécifier un motif qui varie à chaque passage. (Pour faire une compilation une seule fois lors de l'exécution, utilisez /\$variable/o.)

Un cas spécial : spécifier un blanc ( ' ') comme MOTIF découpe selon les espaces exactement comme le fait split() sans argument. Donc, split(' ') peut être utilisé pour émuler le comportement par défaut de **awk** alors que split(/ /) vous donnera autant de champs vides que d'espaces au début. Un split avec /\s+/ est comme split(' ') sauf dans le cas de blancs au début qui produiront un premier champ vide. Un split sans argument effectue réellement un split(' ', \$\_) en interne.

Un MOTIF /^/ sera traité comme si c'était /^/m sinon il ne serait d'aucune utilité.

Exemple :

```
open(PASSWD, '/etc/passwd');
while (<PASSWD>) {
 chomp;
 ($login, $passwd, $uid, $gid,
 $gcos, $home, $shell) = split(/:/);
 #...
}
```

Comme dans le cas de la reconnaissance d'expressions rationnelles, tout sous-groupe mémorisé non reconnue durant le split() produira un undef :

```
@fields = split /(A)|B/, "1A2B3";
@fields est (1, 'A', 2, undef, 3)
```

### sprintf FORMAT, LISTE

Retourne une chaîne formatée selon les conventions usuelles de la fonction sprintf() de la bibliothèque C. Voir ci-dessous et printf(3) ou printf(3) sur votre système pour une explication sur les principes généraux.

Par exemple :

```
Produire un nombre avec jusqu'à 8 zéros devant
$result = sprintf("%08d", $number);

Arrondir un nombre à la troisième décimale
$rounded = sprintf("%.3f", $number);
```

Perl a sa propre implémentation de sprintf() – elle émule la fonction C sprintf() mais elle ne l'utilise pas (sauf pour les nombres en virgule flottante, et encore en n'autorisant que les modificateurs standards). Conséquence : une extension non standard de votre version locale de sprintf() ne sera pas disponible en Perl.

Au contraire de printf, sprintf ne fera probablement pas ce que vous voulez si vous lui passez un tableau comme premier argument. Ce tableau sera évalué dans un contexte scalaire et, au lieu d'utiliser le premier élément du tableau comme FORMAT, Perl utilisera le nombre d'éléments de ce tableau comme FORMAT, ce qui n'est pas vraiment utilisable.

Le sprintf() de Perl autorise les conversions universellement connues :

```

%% un signe pourcent
%c un caractère dont on fournit le code
%s une chaîne
%d un entier signé, en décimal
%u un entier non-signé, en décimal
%o un entier non-signé, en octal
%x un entier non-signé, en hexadécimal
%e un nombre en virgule flottante, en notation scientifique
%f un nombre en virgule flottante, avec un nombre de décimales fixe
%g un nombre en virgule flottante, %e ou %f (au mieux)

```

De plus, Perl autorise les conversions largement supportées :

```

%X comme %x mais avec des lettres majuscules
%E comme %e, mais en utilisant un "E" majuscule
%G comme %g, mais en utilisant un "E" majuscule (si nécessaire)
%b un entier non signé, en binaire
%p un pointeur (affiche la valeur Perl de l'adresse en hexadécimal)
%n spécial: stocke le nombre de caractères produits dans la
prochaine variable de la liste des paramètres

```

Et finalement, pour des raisons de compatibilité (et "uniquement" pour cela), Perl autorise les conversions inutiles mais largement supportées :

```

%i un synonyme de %d
%D un synonyme de %ld
%U un synonyme de %lu
%O un synonyme de %lo
%F un synonyme de %f

```

Notez que le nombre de chiffres utilisés pour l'exposant en notation scientifique produit par %e, %E, %g et %G lorsque cet exposant est inférieur à 100 dépend du système : ça peut être 3 chiffres ou moins (avec d'éventuels zéros initiaux). En d'autres termes, 1,23 multiplié par 10 à la puissance 99 peut être "1.23e99" ou "1.23e099".

Entre le % et la lettre de format, vous pouvez spécifier un certain nombre d'attributs supplémentaires qui permettent de contrôler l'interprétation du format. Dans l'ordre, on trouve :

#### un index de choix de paramètre

Un index de choix explicite de paramètre tel que %2\$. Par défaut, sprintf formatera le prochain argument inutilisé de la liste mais cela vous permet de choisir votre argument sans respecter cet ordre. Par exemple :

```

printf '%2$d %1$d', 12, 34; # affiche "34 12"
printf '%3$d %d %1$d', 1, 2, 3; # affiche "3 1 1"

```

#### des flags

un ou plusieurs flags parmi :

```

espace précède les nombres positifs par un espace
+ précède les nombres positifs par un signe plus
- justifie le champ à gauche
0 utilise des zéros à la place des espaces pour justifier à droite
précède le nombre non nul en octal par "0",
en hexadécimal par "0x" et en binaire par "0b"

```

Par exemple :

```

printf '<% d>', 12; # affiche "< 12>"
printf '<+%d>', 12; # affiche "<+12>"
printf '<%6s>', 12; # affiche "< 12>"
printf '<%-6s>', 12; # affiche "<12 >"
printf '<%06s>', 12; # affiche "<000012>"
printf '<%#x>', 12; # affiche "<0xc>"

```

#### des flags vectoriels

Ce flag indique à perl d'interpréter le chaîne fournie comme un vecteur d'entiers, un pour chaque caractère de la chaîne. Perl applique le format sur chacun des entier, puis concatène les chaînes obtenues en intercalant un séparateur entre eux (un point . par défaut). C'est pratique pour afficher des valeurs ou des caractères ordinaux dans des chaînes quelconques :

```
printf "%vd", "AB\x{100}"; # affiche "65.66.256"
printf "version is v%vd\n", $^V; # La version de Perl
```

Placez un astérisque `*` avant le `v` pour changer la chaîne utilisée comme séparateur de nombres :

```
printf "address is %*vX\n", ":", $addr; # adresse IPv6
printf "bits are %0*v8b\n", " ", $bits; # une chaîne de bits quelconque
```

Vous pouvez aussi spécifier explicitement le numéro d'argument à utiliser en tant que séparateur :

```
printf '%*4$vX %*4$vX %*4$vX', @addr[1..3], ":"; # 3 adresses IPv6
```

### largeur (minimale)

Les arguments sont habituellement formatés pour utiliser juste la place nécessaire à la représentation de la valeur. Vous pouvez forcer la largeur en plaçant directement un nombre ou en obtenant la largeur depuis l'argument suivant (via `*`) ou même depuis un argument spécifique (via `*2$`) :

```
printf '<%s>', "a"; # affiche "<a>"
printf '<%6s>', "a"; # affiche "< a>"
printf '<%*s>', 6, "a"; # affiche "< a>"
printf '<%*2$s>', "a", 6; # affiche "< a>"
printf '<%2s>', "long"; # affiche "<long>" (ne tronque pas)
```

Si la largeur obtenue par `*` est négative, cela a le même effet que le flag `-` : une justification à gauche.

### précision, ou largeur maximale

Vous pouvez préciser la précision (pour les conversions numériques) ou la largeur maximale (pour les conversions de chaînes) en spécifiant un `.` suivi d'un nombre. Pour les nombres en virgule flottante, à l'exception de `'g'` et `'G'`, cela indique le nombre de décimales à afficher (par défaut 6). Exemple :

```
ces exemples peuvent varier selon le système
printf '<%f>', 1; # affiche "<1.000000>"
printf '<%.1f>', 1; # affiche "<1.0>"
printf '<%.0f>', 1; # affiche "<1>"
printf '<%e>', 10; # affiche "<1.000000e+01>"
printf '<%.1e>', 10; # affiche "<1.0e+01>"
```

Pour `'g'` et `'G'`, cela indique le nombre maximum de chiffres à montrer qu'ils soient avant ou après la virgule. Exemples :

```
ces exemple peuvent changer selon les spécificités du système
printf '<%g>', 1; # affichera "<1>"
printf '<%.10g>', 1; # affichera "<1>"
printf '<%g>', 100; # affichera "<100>"
printf '<%.1g>', 100; # affichera "<1e+02>"
printf '<%.2g>', 100.01; # affichera "<1e+02>"
printf '<%.5g>', 100.01; # affichera "<100.01>"
printf '<%.4g>', 100.01; # affichera "<100>"
```

Pour les nombres entiers, la spécification d'une précision indique la largeur du nombre à afficher. Il sera éventuellement précédé de zéros pour atteindre cette largeur :

```
printf '<%.6x>', 1; # affiche "<000001>"
printf '<%#.6x>', 1; # affiche "<0x000001>"
printf '<%-10.6x>', 1; # affiche "<000001 >"
```

Pour les conversions de chaînes, la spécification d'une précision tronquera la chaîne pour qu'elle tienne dans cette longueur maximale :

```
printf '<%.5s>', "truncated"; # affiche "<trunc>"
printf '<%10.5s>', "truncated"; # affiche "< trunc>"
```

Vous pouvez aussi récupérer la valeur de précision dans le prochain argument via `.*` :

```
printf '<%.6x>', 1; # affiche "<000001>"
printf '<%.*x>', 6, 1; # affiche "<000001>"
```

Ce n'est pas encore faisable mais une future version proposera de récupérer la précision dans un argument choisi en utilisant par exemple `.*2$` :

```
printf '<%.*2$x>', 1, 6; # INVALIDE, mais affichera un jour "<000001>"
```

### taille

Pour les conversions numériques, vous pouvez préciser la taille à utiliser pour interpréter le nombre en utilisant `l`, `h`, `V`, `q`, `L` ou `ll`. Pour les conversions d'entiers (`d u o x X b i D U O`), on suppose que la taille est celle utilisée par défaut par votre système pour un entier (habituellement 32 ou 64 bits). Mais vous pouvez changer cela en précisant vous-même un type standard C (l'un de ceux connus du compilateur ayant compilé Perl) à utiliser à la place :

```
l interprète un entier comme étant
 du type C "long" ou "unsigned long"
h interprète un entier comme étant
 du type C "short" ou "unsigned short"
q, L ou ll interprète un entier comme étant
 du type C "long long", "unsigned long long" ou
 "quads" (habituellement des entiers 64 bits)
```

Ce dernier type produira une erreur si la version de Perl installée ne comprend pas les "quads". (Pour qu'ils soient reconnus il faut soit une plateforme qui les connaisse nativement soit une version de Perl compilée spécifiquement pour les reconnaître.) Vous pouvez vérifier que votre version de Perl accepte les quads via *Config* :

```
use Config;
($Config{use64bitint} eq 'define' || $Config{longsize} >= 8) &&
 print "quads\n";
```

Pour les conversions de nombres à virgule flottante (`e f g E F G`), on suppose que le nombre est du type utilisé par défaut sur votre système (double ou long double). Mais vous pouvez contraindre l'usage des 'long doubles' grâce à `q`, `L` ou `ll` si votre plateforme connaît ces types. Vous pouvez vérifier que votre version de Perl accepte les 'long double' via *Config* :

```
use Config;
$Config{d_longdbl} eq 'define' && print "long doubles\n";
```

Vous pouvez aussi tester si votre installation de Perl considère que 'long double' est le type par défaut pour les nombres à virgule flottante via *Config* :

```
use Config;
($Config{uselongdouble} eq 'define') &&
 print "long doubles by default\n";
```

Il y a aussi le cas où les 'long doubles' et les 'doubles' désignent la même chose :

```
use Config;
($Config{doublesize} == $Config{longdblsize}) &&
 print "doubles are long doubles\n";
```

Le spécificateur de taille `V` n'a aucun effet pour le code Perl mais il est accepté pour des raisons de compatibilité avec du code XS ; il signifie « utilise la taille standard pour un entier (ou un nombre en virgule flottante) Perl », ce qui est déjà le cas par défaut dans le code Perl.

### l'ordre des arguments

Normalement, `sprintf` utilise le premier argument encore inutilisé comme valeur à formater pour chaque spécification de format. Si cette spécification utilise `*` pour utiliser des arguments supplémentaires, ils sont pris dans la liste d'arguments dans l'ordre où ils apparaissent dans la spécification *avant* la valeur elle-même. Lorsqu'un argument est spécifié en utilisant un index explicite, cela ne modifie en rien l'ordre d'utilisation normal des arguments.

Donc :

```
printf '<%.*.*s>', $a, $b, $c;
```

devrait utiliser `$a` pour la largeur, `$b` pour la précision et `$c` comme valeur à formater alors que :

```
print '<.*1$.*s>', $a, $b;
```

devrait utiliser `$a` comme largeur et comme précision et `$b` comme valeur à formater.

Voici quelques exemples supplémentaires - notez bien que pour utiliser les index explicites, il faut parfois protéger le `$` :

```
printf "%2\$d %d\n", 12, 34; # affichera "34 12\n"
printf "%2\$d %d %d\n", 12, 34; # affichera "34 12 34\n"
printf "%3\$d %d %d\n", 12, 34, 56; # affichera "56 12 34\n"
printf "%2\$*3\$d %d\n", 12, 34, 3; # affichera " 34 12\n"
```

Si `use locale` est actif, le caractère utilisé comme séparateur décimal pour les nombres réels dépend de la valeur de `LC_NUMERIC`. Voir *perllocale*.

### sqrt EXPR

#### sqrt

Renvoie la racine carrée de `EXPR`. Si `EXPR` est omis, retourne la racine carrée de `$_`. Ne fonctionne que sur les opérandes non négatifs à moins que vous n'ayez chargé le module standard `Math::Complex`.

```
use Math::Complex;
print sqrt(-2); # affiche 1.4142135623731i
```

### srand EXPR

#### srand

Fixe la graine aléatoire pour l'opérateur `rand()`.

La seule utilité de cette fonction est d'initialiser la fonction `rand` pour que cette dernière produise une séquence différente à chaque exécution de votre programme.

Si `srand()` n'est pas appelé explicitement, il l'est implicitement lors de la première utilisation de l'opérateur `rand()`. Par contre, ce n'était pas le cas dans les versions de Perl antérieures à la version 5.004 et donc, si votre script doit pouvoir tourner avec de vieilles versions de Perl, il doit appeler `srand()`.

La plupart des programmes n'ont pas besoins d'appeler `srand()`. Seuls le feront ceux ayant besoin d'une graine cryptographiquement forte puisqu'ils ne peuvent se satisfaire de la valeur par défaut calculée en fonction de l'heure courante, du numéro de processus et de l'allocation mémoire ou en fonction du device `/dev/urandom` lorsqu'il est disponible.

Vous pouvez appeler `srand($graine)` avec la même `$graine` pour que `rand()` reproduise la *même* séquence mais cela est réservé à la production de données prévisibles afin de faciliter les tests et le débogage.

N'appellez **pas** `srand()` (sans argument) plus d'une fois par exécution de votre script. L'état interne du générateur de nombres aléatoires devrait contenir plus d'entropie que celle fournie par une graine quelconque. Donc un nouvel appel à `srand()` ne peut qu'amener à *perdre* de l'aléa.

La plupart des implémentations de `srand` utilise un entier et ignoreront silencieusement une éventuelle partie décimale. Cela signifie que l'appel `srand(42)` produira le même résultat que l'appel `srand(42.1)`. Pour être correct, passez toujours un entier à `srand`.

Dans les versions de Perl antérieures à la version 5.004, la valeur par défaut était juste `time()`. Ce n'est pas une graine particulièrement bonne et donc de nombreux programmes anciens fournissaient leur propre valeur de graine (souvent `time ^ $$` ou `time ^ ($$ + ($$ << 15))`) mais ce n'est plus nécessaire maintenant.

En revanche, pour des applications cryptographiques, vous devez utiliser une graine bien plus aléatoire que celle par défaut. Le checksum d'une sortie compressée d'un ou plusieurs programmes systèmes dont les valeurs changent rapidement est une méthode usuelle. Par exemple :

```
srand (time ^ $$ ^ unpack "%L*", 'ps axww | gzip');
```

Si cela vous intéresse tout particulièrement, regardez le module CPAN `Math::TrulyRandom`.

Les programmes fréquemment utilisés (comme des scripts CGI) qui utilisent simplement :

```
time ^ $$
```

comme graine peuvent tomber sur la propriété mathématique suivante :

$$a^b == (a+1)^{(b+1)}$$

une fois sur trois. Donc ne faites pas ça.

### stat DESCRIPTEUR

#### stat EXPR

#### stat

Retourne une liste de 13 éléments donnant des informations sur l'état soit du fichier dont le nom est donné par `EXPR` soit du fichier ouvert par `DESCRIPTEUR`. Si `EXPR` est omis, retourne l'état de `$_`. Retourne une liste vide en cas d'échec. Typiquement utilisé de la manière suivante :

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
 $atime,$mtime,$ctime,$blksize,$blocks)
 = stat($filename);
```

Certains champs ne sont pas gérés par certains types de systèmes de fichiers. Voici la signification de ces champs :

```
0 dev numéro de device du système de fichiers
1 ino numéro d'inode
2 mode droits du fichier (type et permissions)
3 nlink nombre de liens (hard) sur le fichier
4 uid ID numérique de l'utilisateur propriétaire du fichier
5 gid ID numérique du groupe propriétaire du fichier
6 rdev l'identificateur de device (fichiers spéciaux uniquement)
7 size taille totale du fichier, en octets
8 atime date de dernier accès en secondes depuis l'origine des temps
9 mtime date de dernière modification en secondes depuis
 l'origine des temps
10 ctime date de dernière modification de l'inode en secondes
 depuis l'origine des temps (*)
11 blksize taille de blocs préférée pour les E/S sur fichiers
12 blocks nombre de blocs réellement occupés
```

(Sur la plupart des systèmes, l'origine des temps est fixée au 1er janvier 1970 à minuit GMT.)

(\*) Ces champs n'existent pas obligatoirement sur tous les systèmes de fichiers. Par exemple, le champ `ctime` n'est pas portable. En particulier, ne comptez pas dessus pour représenter une « date de création ». Voir *Fichiers in perlport* pour plus de détails.

Si vous passez à `stat` le descripteur spécial dont le nom est le caractère souligné seul, aucun appel à `stat` n'est effectué par contre le contenu courant de la structure d'état du dernier appel à `stat`, à `lstat` ou du dernier test de fichier est retourné. Exemple :

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
 print "$file is executable NFS file\n";
}
```

(Ceci ne marche que sur les machines dont le numéro de device est négatif sous NFS.)

Comme le mode contient à la fois le type de fichier et les droits d'accès, vous devrez masquer la portion concernant le type de fichier et utiliser `(s)printf` avec le format `"%o"` pour voir les véritables permissions :

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

Dans un contexte scalaire, `stat()` retourne une valeur booléenne indiquant le succès ou l'échec et positionne, en cas de succès, les informations liées au descripteur spécial `_`.

Le module `File::stat` fournit un mécanisme pratique d'accès par nom :

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
 $filename, $sb->size, $sb->mode & 07777,
 scalar localtime $sb->mtime;
```

Vous pouvez importer les constantes symboliques de permissions (`S_IF*`) et les fonctions de test (`S_IS*`) depuis le module `Fcntl` :

```
use Fcntl ':mode';

$mode = (stat($filename))[2];

$user_rwx = ($mode & S_IRWXU) >> 6;
$group_read = ($mode & S_IRGRP) >> 3;
$other_exec = $mode & S_IXOTH;

printf "Permissions are %04o\n", S_IMODE($mode), "\n";

$is_setuid = $mode & S_ISUID;
$is_setgid = S_ISDIR($mode);
```



Vous pourriez écrire ces deux derniers exemples en utilisant les opérateurs `-u` et `-d`. Les constantes communes `S_IF*` disponibles sont :

```
Droits : lecture, écriture, exécution,
pour utilisateur, groupe, autres.

S_IRWXU S_IRUSR S_IWUSR S_IXUSR
S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH

Setuid/Setgid/Stickness/SaveText.
Leur sémantique exacte dépend du système

S_ISUID S_ISGID S_ISVTX S_ISTXT

File types. Not necessarily all are available on your system.

S_IFREG S_IFDIR S_IFLNK S_IFBLK S_IFCHR S_IFIFO S_IFSOCK S_IFWHT S_ENFMT

The following are compatibility aliases for S_IRUSR, S_IWUSR, S_IXUSR.

S_IREAD S_IWRITE S_IEXEC
```

et les fonctions `S_IF*` <sont :>

```
S_IME($mode) the part of $mode containing the permission bits
 and the setuid/setgid/sticky bits

S_IFMT($mode) the part of $mode containing the file type
 which can be bit-anded with e.g. S_IFREG
 or with the following functions

Les opérateurs -f, -d, -l, -b, -c, -p, and -S.

S_ISREG($mode) S_ISDIR($mode) S_ISLNK($mode)
S_ISBLK($mode) S_ISCHR($mode) S_ISFIFO($mode) S_ISSOCK($mode)

No direct -X operator counterpart, but for the first one
the -g operator is often equivalent. The ENFMT stands for
record flocking enforcement, a platform-dependent feature.

S_IENFMT($mode) S_ISWHT($mode)
```

Voir la documentation native de `chmod(2)` et de `stat(2)` pour de meilleures informations au sujet des constantes `S_*`. Pour obtenir des informations sur un lien symbolique au lieu du fichier vers lequel il pointe, utilisez la fonction `lstat`.

## study SCALAIRE

### study

Prend du temps supplémentaire pour étudier (`study`) la chaîne SCALAIRE (ou `$_` si SCALAIRE est omis) afin d'anticiper de nombreuses recherches d'expressions rationnelles sur elle avant sa prochaine modification. Cela peut améliorer ou non le temps de recherche selon la nature et le nombre de motifs que vous recherchez et selon la fréquence de distribution des caractères dans la chaîne – vous devriez probablement comparer les temps d'exécution avec et sans pour savoir quand cela est plus rapide. Les boucles qui recherchent de nombreuses petites chaînes constantes (même celles comprises dans des motifs plus complexes) en bénéficient le plus. Il ne peut y avoir qu'un seul `study()` actif à la fois – si vous étudiez un autre scalaire, le précédent est "oublié". (`study()` fonctionne de la manière suivante : on construit une liste chaînée de tous les caractères de la chaîne à étudier ce qui permet de savoir, par exemple, où se trouve tous les 'k'. Dans chaque chaîne recherchée, on choisit le caractère le plus rare en se basant sur une table de fréquences construite à partir de programmes C et de textes anglais. Seuls sont examinés les endroits qui contiennent ce caractère "rare".)

Par exemple : voici la boucle qui insère une entrée d'index devant chaque ligne contenant un certain motif :

```
while (<>) {
 study;
 print ".IX foo\n" if /\bfoo\b/;
 print ".IX bar\n" if /\bbar\b/;
 print ".IX blurfl\n" if /\bblurfl\b/;
 # ...
 print;
}
```

Lors de la recherche de `/\bfoo\b/`, seuls sont examinés les endroits de `$_` contenant `f` parce que `f` est plus rare que `o`. En général, le gain est important sauf dans des cas pathologiques. Savoir si l'étude initiale est moins coûteuse que le temps gagné lors de la recherche est la seule vraie question.

Remarquez que si vous faites une recherche sur des chaînes que vous ne connaissez que lors de l'exécution, vous pouvez alors construire une boucle entière dans une chaîne que vous évaluerez via `eval()` afin d'éviter de recompiler vos motifs à chaque passage. Combiné avec l'affectation de `undef` à `$/` pour lire chaque fichier comme un seul enregistrement, cela peut être extrêmement rapide et parfois même plus rapide que des programmes spécialisés comme `fgrep(1)`. Le code suivant recherche une liste de mots (`@words`) dans une liste de fichiers (`@files`) et affiche la liste des fichiers qui contiennent ces mots :

```
$search = 'while (<>) { study;';
foreach $word (@words) {
 $search .= "++\$seen{\$ARGV} if /\b$word\b;\n";
}
$search .= "};";
@ARGV = @files;
undef $/;
eval $search; # ca parle...
$/ = "\n"; # retour au délimiteur de ligne normal
foreach $file (sort keys(%seen)) {
 print $file, "\n";
}
}
```

#### sub NOM BLOC

#### sub NOM (PROTO) BLOC

#### sub NOM : ATTRS BLOC

#### sub NOM (PROTO) : ATTRS BLOC

C'est la définition de subroutine. Pas vraiment une fonction en tant que telle. Avec juste le nom `NOM` (et un éventuel prototype), c'est une simple déclaration préalable. Sans `NOM`, c'est la déclaration d'une fonction anonyme et cela retourne une valeur : la référence du `CODE` de la fermeture que vous venez de créer.

Voir *perlsb* et *perlref* pour plus de détails sur les subroutines et les références et *attributes* et *Attribute::Handlers* pour plus de détails au sujet des attributs (`ATTRS`).

#### substr EXPR,OFFSET,LONGUEUR,REMPLACEMENT

#### substr EXPR,OFFSET,LONGUEUR

#### substr EXPR,OFFSET

Extrait une sous-chaîne de `EXPR` et la retourne. Le premier caractère est à l'indice `0` ou à ce que vous avez fixé par `$[` (mais ne le faites pas). Si `OFFSET` est négatif (ou plus précisément plus petit que `$[`), le compte a lieu à partir de la fin de la chaîne. Si `LONGUEUR` est omis, retourne tous les caractères jusqu'à la fin de la chaîne. Si `LONGUEUR` est négatif, il indique le nombre de caractères à laisser à la fin de la chaîne.

Vous pouvez utiliser la fonction `substr()` comme une lvalue auquel cas `EXPR` doit aussi être une lvalue. Si vous affectez quelque chose de plus court que `LONGUEUR`, la chaîne raccourcit et si vous affectez quelque chose de plus long que `LONGUEUR`, la chaîne grossit. Pour conserver la même longueur vous pouvez remplir ou couper votre valeur en utilisant `sprintf()`.

Si `OFFSET` et `LONGUEUR` spécifient une sous-chaîne qui est partiellement en dehors de la chaîne, seule la partie qui est dans la chaîne qui est retournée. Si la sous-chaîne est entièrement en dehors de la chaîne, un message d'avertissement (warning) est produit et la valeur `undef` est retournée. Lorsque `substr()` est utilisé en tant que lvalue, spécifier une sous-chaîne entièrement en dehors de la chaîne produit une erreur fatale. Voici un exemple illustrant ce comportement :

```
my $name = 'fred';
substr($name, 4) = 'dy'; # $name vaut maintenant 'freddy'
my $null = substr $name, 6, 2; # retourne '' (sans avertissement)
my $oops = substr $name, 7; # retourne undef, avec avertissement
substr($name, 7) = 'gap'; # erreur fatale
```

Un autre moyen d'utiliser `substr()` comme lvalue est de spécifier la chaîne de remplacement comme quatrième argument (`REMPLACEMENT`). Ceci permet de remplacer une partie de la chaîne en récupérant ce qui y était auparavant en une seule opération exactement comme avec `splice()`.

**symlink OLDFILE,NEWFILE**

Crée un nouveau fichier (NEWFILE) lié symboliquement au vieux fichier (OLDFILE). Retourne 1 en cas de succès ou 0 autrement. Produit une erreur fatale lors de l'exécution sur les systèmes qui ne supportent pas les liens symboliques. Pour vérifier cela, utilisez eval :

```
$symlink_exists = eval { symlink("", ""); 1 };
```

**syscall LISTE**

Réalise l'appel système spécifié comme premier élément de la liste en lui passant les éléments restants comme arguments. Cela produit une erreur fatale s'il n'est pas implémenté. Les arguments sont interprétés de la manière suivante : si un argument donné est numérique, il est passé comme un entier. Sinon, c'est le pointeur vers la valeur de la chaîne qui est passé. Vous avez la charge de vous assurer qu'une chaîne est déjà assez longue pour recevoir un résultat qui pourrait y être écrit. Vous ne pouvez pas utiliser de chaîne littérale (ou autres chaînes en lecture seule) comme argument de syscall() parce que Perl s'assure qu'il est possible d'écrire dans toutes les chaînes dont il passe les pointeurs. Si votre argument numérique n'est pas un littéral et n'a jamais été interprété dans un contexte numérique, vous pouvez lui ajouter 0 pour forcer Perl à le voir comme un nombre. Le code suivant émule la fonction syswrite() :

```
require 'syscall.ph'; # peut nécessiter de faire tourner h2ph
$s = "hi there\n";
syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Remarquez que Perl ne peut pas passer plus de 14 arguments à votre appel système, ce qui en pratique suffit largement.

Syscall retourne la valeur retournée par l'appel système appelé. Si l'appel système échoue, syscall() retourne -1 et positionne \$! (errno). Remarquez que certains appels systèmes peuvent légitimement retourner -1. Le seul moyen de gérer cela proprement est de faire \$!=0 avant l'appel système et de regarder la valeur de \$! lorsque syscall retourne -1.

Il y a un problème avec syscall(&SYS\_pipe) : cela retourne le numéro du fichier créé côté lecture du tube. Il n'y a aucun moyen de récupérer le numéro de fichier de l'autre côté. Vous pouvez contourner ce problème en utilisant pipe() à la place.

**sysopen DESCRIPTEUR,FILENAME,MODE****sysopen DESCRIPTEUR,FILENAME,MODE,PERMS**

Ouvre le fichier dont le nom est donné par FILENAME et l'associe avec DESCRIPTEUR. Si DESCRIPTEUR est une expression, sa valeur représente le nom du véritable descripteur à utiliser. Cette fonction appelle la fonction open() du système sous-jacent avec les paramètres FILENAME, MODE et PERMS.

Les valeurs possibles des bits du paramètre MODE sont dépendantes du système; elles sont disponibles via le module standard Fcntl. Lisez la documentation du open de votre système d'exploitation pour savoir quels sont les bits disponibles. Vous pouvez combiner plusieurs valeurs en utilisant l'opérateur |.

Quelques-unes des valeurs les plus courantes sont O\_RDONLY pour l'ouverture en lecture seule, O\_WRONLY pour l'ouverture en écriture seule, et O\_RDWR pour l'ouverture en mode lecture/écriture.

Pour des raisons historiques, quelques valeurs fonctionnent sur la plupart des systèmes supportés par perl : zéro signifie lecture seule, un signifie écriture seule et deux signifie lecture/écriture. Nous savons que ces valeurs *ne* fonctionnent *pas* sous Unix OS/390 ou sur Macintosh; vous ne devriez sans doute pas les utiliser dans du code nouveau.

Si le fichier nommé FILENAME n'existe pas et que l'appel à open() le crée (typiquement parce que MODE inclut le flag O\_CREAT) alors la valeur de PERM spécifie les droits de ce nouveau fichier. Si vous avez omis l'argument PERM de sysopen(), Perl utilise la valeur octale 0666. Les valeurs de droits doivent être fournies en octal et sont modifiées par la valeur courante du umask de votre processus.

Sur la plupart des systèmes, le flag O\_EXCL permet d'ouvrir un fichier en mode exclusif. Ce n'est **pas** du verrouillage : l'exclusivité signifie ici que si le fichier existe déjà, sysopen() échoue. O\_EXCL peut ne pas marcher sur des systèmes de fichiers réseaux et n'a aucun effet à moins que O\_CREAT soit présent. Utiliser O\_CREAT|O\_EXCL empêche le fichier d'être ouvert si c'est un lien symbolique. Cela ne protège pas des liens symboliques dans le chemin.

Parfois vous voudrez tronquer un fichier existant. C'est faisable via O\_TRUNC. Le comportement de O\_TRUNC avec O\_RDONLY n'est pas défini.

Vous devriez éviter d'imposer un mode 0644 comme argument de sysopen parce que cela enlève à l'utilisateur la possibilité de fixer un umask plus permissif. Voir umask pour plus de détails.

Notez que sysopen dépend de la fonction fdopen() de votre bibliothèque C. Sur de nombreux systèmes UNIX, fdopen() est connue pour échouer si le nombre de descripteurs de fichiers excède une certaine valeur, typiquement

255. Si vous avez besoin de plus de descripteurs, pensez à recompiler Perl en utilisant la bibliothèque `sfio` ou à utiliser la fonction `POSIX::open()`.

Voir *perlopentut* pour une initiation à l'ouverture de fichiers.

### **sysread** DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET

#### **sysread** DESCRIPTEUR,SCALAIRE,LONGUEUR

Tente de lire `LONGUEUR` octets de données à partir du `DESCRIPTEUR` spécifié en utilisant l'appel système `read(2)` pour les stocker dans la variable `SCALAIRE`. Comme cette fonction n'utilise pas les entrées/sorties bufferisées, le mélange avec d'autres sortes de lecture/écriture comme `print`, `write`, `seek`, `tell` ou `eof` peut mal se passer parce que habituellement `PerlIO` ou `stdio` bufferise les données. Retourne le nombre d'octets réellement lus, `0` à la fin du fichier ou `undef` en cas d'erreur (dans ce dernier cas `$!` est défini). `SCALAIRE` grossira ou diminuera afin que le dernier octet lu soit effectivement le dernier octet du scalaire après la lecture.

Un `OFFSET` peut être spécifié pour placer les données lues ailleurs qu'au début de la chaîne. Un `OFFSET` négatif spécifie un emplacement en comptant les caractères à partir de la fin de la chaîne. Un `OFFSET` positif plus grand que la longueur de `SCALAIRE` agrandira la chaîne jusqu'à la taille requise en la remplissant avec des octets `"\0"` avant de lui ajouter le résultat de la lecture.

Il n'y a pas de fonction `syseof()`, ce qui n'est pas un mal puisque `eof()` ne marche pas très bien sur les fichiers device (comme les tty). Utilisez `sysread()` et testez une valeur de retour à zéro pour savoir si c'est terminé.

Notez que si le `DESCRIPTEUR` a été marqué comme `:utf8`, ce sont des caractères Unicode qui sont lus à la place des octets (le `LONGUEUR`, l'`OFFSET` et la valeur retournée par `sysread()` sont exprimés en terme de caractères Unicode). Les filtres `:encoding(...)` introduisent implicitement le filtre `:utf8`. Voir `binmode`, `open` et la directive `open` dans *open*.

#### **sysseek** DESCRIPTEUR,POSITION,WHENCE

Spécifie, en octets, la position système d'un `DESCRIPTEUR` en utilisant l'appel système `lseek(2)`. `DESCRIPTEUR` peut être une expression qui donne le nom du descripteur à utiliser. Les valeurs de `WHENCE` sont `0` pour mettre la nouvelle position à `POSITION`, `1` pour la mettre à la position courante plus `POSITION` et `2` pour la mettre à EOF plus `POSITION` (typiquement une valeur négative).

Notez bien « en octets » : même si le `DESCRIPTEUR` a été configuré pour opérer sur des caractères (en utilisant le filtre d'entrée/sortie `:utf8` par exemple), `tell()` retournera une position en octets et non en caractères (l'implémentation d'une telle fonctionnalité aurait ralenti énormément `sysseek()`).

`sysseek()` passe outre les tampons habituels d'entrée/sortie. Donc le mélange avec d'autres sortes de lecture/écriture (autre que `sysread()`) comme `<>`, `read()`, `print()`, `write()`, `seek()` ou `tell()` peut tout casser.

Pour `WHENCE`, vous pouvez utiliser les constantes `SEEK_SET`, `SEEK_CUR` et `SEEK_END` (début de fichier, position courante, fin de fichier) provenant du module `Fcntl`. L'usage de ces constantes rend votre script plus portable. Voici un exemple qui définit la fonction "systemell" :

```
use Fcntl 'SEEK_CUR';
sub systemell { sysseek($_[0], 0, SEEK_CUR) }
```

Retourne la nouvelle position ou `undef` en cas d'échec. Une position nulle est retournée par la valeur `"0 but true"`; Donc `sysseek()` retourne `true` (vrai) en cas de succès et `false` (faux) en cas d'échec et vous pouvez encore déterminer facilement la nouvelle position.

### **system** LISTE

#### **system** PROGRAMME LISTE

Fait exactement la même chose que `exec LISTE` sauf qu'un `fork` est effectué au préalable et que le process parent attend que son fils ait terminé. Remarquez que le traitement des arguments dépend de leur nombre. Si il y a plus d'un argument dans `LISTE` ou si `LISTE` est un tableau avec plus d'une valeur, `system` exécute le programme donné comme premier argument avec comme arguments ceux donnés dans le reste de la liste. Si il n'y a qu'un seul argument dans `LISTE` et s'il contient des méta-caractères du shell, il est passé en entier au shell de commandes du système pour être interprété (c'est `/bin/sh -c` sur les plates-formes Unix mais cela peut varier sur les autres). Si il ne contient pas de méta-caractères du shell, il est alors découpé en mots et passé directement à `execvp()`, ce qui est plus efficace.

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant d'effectuer une opération impliquant un `fork()` mais cela n'est pas supporté sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionner la variable `$|` (`$AUTOFLUSH` en anglais) ou appelé la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts.

La valeur retournée est le statut de sortie (exit status) du programme tel que retourné par l'appel `wait()`. Pour obtenir la valeur réelle de sortie, il faut décaler la valeur de retour de 8 bits vers la droite. Voir aussi `exec`. Ce n'est pas ce qu'il faut utiliser pour capturer la sortie d'une commande. Pour cela, regarder les apostrophes inversées

(backticks) ou `qx//` comme décrit dans 'CHAINE' in *perlop*. Une valeur de retour -1 indique l'échec du lancement du programme ou une de l'appel système `wait(2)` (\$! en donne la raison).

Comme `exec()`, `system()` vous autorise à définir le nom sous lequel le programme apparaît si vous utilisez la syntaxe "system PROGRAMME LISTE". Voir `exec`.

Comme les signaux SIGINT et SIGQUIT sont ignorés durant l'exécution de `system`, si vous en avez besoin, vous devrez vous débrouiller pour les gérer indirectement en vous basant sur la valeur de retour.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
 or die "system @args failed: $?"
```

Vous pouvez tester tous les cas possibles d'échec en analysant \$? de la manière suivante :

```
if ($? == -1) {
 print "failed to execute: $!\n";
}
elsif ($? & 127) {
 printf "child died with signal %d, %s coredump\n",
 ($? & 127), ($? & 128) ? 'with' : 'without';
}
else {
 printf "child exited with value %d\n", $? >> 8;
}
```

ou, de manière plus portable, en utilisant les appels `W*()` du module POSIX. Voir *perlport* pour plus d'information.

Lorsque les arguments sont exécutés via le shell système, les résultats et codes de retour sont sujet à tous ses caprices et capacités. Voir 'CHAINE' in *perlop* et `exec` pour plus de détails.

**syswrite** DESCRIPTEUR,SCALAIRE,LONGUEUR,OFFSET

**syswrite** DESCRIPTEUR,SCALAIRE,LONGUEUR

**syswrite** DESCRIPTEUR,SCALAIRE

Essaye d'écrire LONGUEUR octets provenant de la variable SCALAIRE sur le DESCRIPTEUR spécifié en utilisant l'appel système `write(2)`. Si LONGUEUR n'est pas spécifiée, c'est le contenu complet de SCALAIRE qui est écrit. Cette fonction n'utilise pas les tampons d'entrée/sortie habituels. Donc le mélange avec d'autres sortes de lecture/écriture (autre que `sysread()`) comme `print()`, `write()`, `seek()`, `tell()` ou `eof` peut mal se passer parce que habituellement `perlio` ou `stdio` stockent les données dans des tampons. Retourne le nombre d'octets réellement écrits ou `undef` en cas d'erreur (dans ce cas la variable d'erreur \$! est renseignée). Si LONGUEUR est plus grande que la quantité de données disponibles dans SCALAIRE après OFFSET, seules les données disponibles sont écrites.

Un OFFSET peut être spécifié pour lire les données à écrire à partir d'autre chose que le début du scalaire. Un OFFSET négatif calcule l'emplacement en comptant les caractères à partir de la fin de la chaîne. Au cas où SCALAIRE est vide, vous pouvez utiliser OFFSET mais uniquement avec la valeur zéro.

Notez que si le DESCRIPTEUR a été marqué comme `:utf8`, ce sont des caractères Unicode qui sont écrits à la place des octets (la LONGUEUR, l'OFFSET et la valeur retournée par `syswrite()` sont exprimés en terme de caractères Unicode encodés en UTF-8). Les filtres `:encoding(...)` introduisent implicitement le filtre `:utf8`. Voir `binmode`, `open` et la directive `open` dans *open*.

**tell** DESCRIPTEUR

**tell**

Retourne la position courante en *octets* de DESCRIPTEUR ou -1 en cas d'erreur. DESCRIPTEUR peut être une expression dont la valeur donne le nom du descripteurs réel. Si DESCRIPTEUR est omis, on utilise le dernier fichier lu.

Notez bien « en *octets* » : même si le DESCRIPTEUR a été configuré pour opérer sur des caractères (en utilisant le filtre d'entrée/sortie `:utf8` par exemple), `tell()` retournera une position en octets et non en caractères (l'implémentation d'une telle fonctionnalité aurait ralenti énormément `tell()`).

Il n'y a pas de fonction `systemtell`. Utilisez `sysseek(FH, 0, 1)` à la place.

N'utilisez pas `tell()` (ou d'autres opérations d'E/S utilisant des tampons) sur des DESCRIPTEURS qui ont été manipulés par `sysread()`, `syswrite()` ou `sysseek()`. En effet, ces fonctions ignorent complètement les tampons.

**telldir** DIRHANDLE

Retourne la position courante du dernier `readdir` effectué sur DIRHANDLE. La valeur retournée peut être fournie à `seekdir` pour accéder à un endroit particulier dans ce répertoire. `telldir` pose les mêmes problèmes que l'appel système correspondant.

**tie VARIABLE,CLASSNAME,LISTE**

Cette fonction relie une variable à une classe d'un package qui fournit l'implémentation de cette variable. VARIABLE est le nom de la variable à relier. CLASSNAME est le nom de la classe implémentant les objets du type correct. Tout argument supplémentaire est passé tel quel à la méthode "new()" de la classe (à savoir TIESCALAR, TIEARRAY ou TIEHASH). Typiquement, ce sont des arguments tels que ceux passés à la fonction C dbm\_open(). L'objet retourné par la méthode "new()" est aussi retourné par la fonction tie() ce qui est pratique si vous voulez accéder à d'autres méthodes de CLASSNAME.

Remarquez que des fonctions telles que keys() et values() peuvent retourner des listes énormes lorsqu'elles sont utilisées sur de gros objets comme des fichiers DBM. Vous devriez plutôt utiliser la fonction each() pour les parcourir. Exemple :

```
print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
 print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

Une classe implémentant une table de hachage devrait définir les méthodes suivantes :

```
TIEHASH classname, LISTE
FETCH this, key
STORE this, key, value
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
SCALAR this
DESTROY this
UNTIE this
```

Une classe implémentant un tableau devrait définir les méthodes suivantes :

```
TIEARRAY classname, LISTE
FETCH this, key
STORE this, key, value
FETCHSIZE this
STORESIZE this, count
CLEAR this
PUSH this, LISTE
POP this
SHIFT this
UNSHIFT this, LISTE
SPlice this, offset, length, LISTE
EXTEND this, count
DESTROY this
UNTIE this
```

Une classe implémentant un descripteur de fichier devrait définir les méthodes suivantes :

```
TIEHANDLE classname, LISTE
READ this, scalar, length, offset
READLINE this
GETC this
WRITE this, scalar, length, offset
PRINT this, LISTE
PRINTF this, format, LISTE
BINMODE this
EOF this
FILENO this
SEEK this, position, whence
TELL this
```

```

OPEN this, mode, LIST
CLOSE this
DESTROY this
UNTIE this

```

Une classe implémentant un scalaire devrait définir les méthodes suivantes :

```

TIESCALAR classname, LISTE
FETCH this
STORE this, value
DESTROY this
UNTIE this

```

Il n'est pas absolument nécessaire d'implémenter toutes les méthodes décrites ci-dessus. Voir *perltie*, *Tie::Hash*, *Tie::Array*, *Tie::Scalar*, et *Tie::Handle*.

Au contraire de `dbmopen()`, la fonction `tie()` n'effectue pas pour vous le 'use' ou le 'require' du module – vous devez le faire vous-même explicitement. Voir les modules *DB\_File* ou *Config* pour des utilisations intéressantes de `tie()`.

Pour de plus amples informations, voir *perltie* et `tied VARIABLE`.

### **tie VARIABLE**

Retourne une référence vers l'objet caché derrière VARIABLE (la même valeur que celle retournée par l'appel `tie()` qui a lié cette variable à un package). Retourne la valeur `undef` si VARIABLE n'est pas lié à un package.

### **time**

Retourne le nombre de secondes écoulées depuis ce que le système considère comme étant l'origine des temps et peut servir comme argument de `gmtime()` et de `localtime()`. Sur la plupart des systèmes, l'origine des temps est le 1er janvier 1970 ; Sauf Mac OS Classic qui utilise comme origine le 1er janvier 1904 à minuit dans le fuseau horaire local.

Pour mesurer le temps avec une meilleure granularité que la seconde, vous pouvez utiliser le module `Time::HiRes` (disponible sur CPAN ou en standard depuis Perl 5.8) ou, si vous avez `gettimeofday(2)`, vous pouvez utiliser l'interface `syscall` de Perl. Voir *perlfaq8* pour les détails.

### **times**

Retourne une liste de quatre éléments donnant le temps utilisateur et système en secondes, pour ce process et pour les enfants de ce process.

```
($user, $system, $cuser, $csystem) = times;
```

Dans un contexte scalaire `times` retourne `$user`.

### **tr///**

L'opérateur de translittération. La même chose que `y///`. Voir *perlop*.

### **truncate DESCRIPTEUR, LONGUEUR**

#### **truncate EXPR, LONGUEUR**

Tronque le fichier ouvert par DESCRIPTEUR ou nommé par EXPR à la taille LONGUEUR spécifiée. Produit une erreur fatale si la troncature n'est pas implémentée sur votre système. Retourne `true` (vrai) en cas de succès ou la valeur `undef` autrement.

Le comportement de `truncate()` est indéfini si LONGUEUR est plus grand que la longueur actuelle du fichier.

### **uc EXPR <conversion, majuscule>**

#### **uc**

Retourne la version en majuscule de EXPR. C'est la fonction interne qui implémente la séquence d'échappement `\U` des chaînes entre guillemets. Respecte le locale `LC_CTYPE` courant si `use locale` est actif. Voir *perllocale* et *perlunicode* pour plus de détails. Ne gère pas la table spécifique aux majuscules apparaissant en début de mot. Voir *ucfirst* pour cela.

Si EXPR est omis, s'applique à `$_`.

#### **ucfirst EXPR**

#### **ucfirst**

Retourne la valeur de EXPR avec le premier caractère en majuscule. C'est la fonction interne qui implémente la séquence d'échappement `\u` des chaînes entre guillemets. Respecte le locale `LC_CTYPE` courant si `use locale` est actif. Voir *perllocale* et *perlunicode* pour plus de détails.

Si EXPR est omis, s'applique à `$_`.

**umask** *EXPR***umask**

Positionne le umask du process à *EXPR* et retourne la valeur précédente. Si *EXPR* est omis, retourne la valeur courante du umask.

Les droits Unix *rwxr-x-* sont représentés par trois ensembles de trois bits ou par trois nombres octaux : **0750** (le 0 initial indique une valeur octale et ne fait pas partie des chiffres). La valeur de *umask* est un tel nombre qui représente les bits de permissions désactivés. Les valeurs de permissions (ou "droits" ou "mode") que vous passez à *mkdir* ou *sysopen* sont modifiées par votre umask. Par exemple, si vous dites à *sysopen* de créer un fichier avec les droits **0777** et si votre umask vaut **0022** alors le fichier sera réellement créé avec les droits **0755**. Si votre umask vaut **0027** (le groupe ne peut écrire ; les autres ne peuvent ni lire, ni écrire, ni exécuter) alors passer **0666** à *sysopen* créera un fichier avec les droits **0640** (**0666** &~ **027** vaut **0640**).

Voici quelques conseils : fournissez un mode de création de **0666** pour les fichiers normaux (dans *sysopen*) et de **0777** pour les répertoires (dans *mkdir*) et les fichiers exécutables. Cela donne la liberté à l'utilisateur de choisir : si il veut des fichiers protégés, il peut choisir un umask de **022** ou **027** ou même le umask particulièrement antisocial **077**. Dans ce domaine, les programmes peuvent rarement (si ce n'est jamais) prendre de meilleures décisions que l'utilisateur. L'exception concerne les fichiers qui doivent être gardés privé : fichiers de mail, fichiers de cookies des navigateurs, fichiers *.rhosts* et autres.

Si *umask(2)* n'est pas implémenté sur votre système et que vous êtes en train de restreindre les droits d'accès pour *vous-même* (i.e.  $(EXPR \& 0700) > 0$ ), cela produit une erreur fatale lors de l'exécution. Si *umask(2)* n'est pas implémenté et que vous ne modifiez pas vos propres droits d'accès, retourne *undef*.

Souvenez-vous que *umask* est un nombre, habituellement donné en octal ; Ce n'est *pas* une chaîne de chiffres en octal. Voir aussi *oct*, si vous disposez d'une chaîne.

**undef** *EXPR***undef**

Donne à *EXPR* la valeur indéfinie (*undef*). *EXPR* doit être une lvalue. À n'utiliser que sur des scalaires, des tableaux (en utilisant @), des tables de hachage (en utilisant %), des sous-routines (en utilisant &) ou des typeglob (en utilisant \*). (Dire *undef \$hash{\$key}* ne fera probablement pas ce que vous espérez sur la plupart des variables prédéfinies ou sur les liste de valeurs DBM. Ne le faites donc pas ; voir *delete*.) Retourne toujours la valeur *undef*. Vous pouvez omettre *EXPR*, auquel cas rien ne sera indéfini mais vous récupérerez encore la valeur *undef* afin, par exemple, de la retourner depuis une sous-routine, de l'affecter à une variable ou de la passer comme argument. Exemples :

```
undef $foo;
undef $bar{'blurfl'}; # À comparer à : delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz; # détruit $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo; # Ignorer la troisième valeur retournée
```

Remarquez que c'est un opérateur unaire et non un opérateur de liste.

**unlink** *LISTE* **unlink****unlink**

Efface une liste de fichiers. Retourne le nombre de fichiers effacés avec succès.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Remarque : *unlink()* n'essayera pas d'effacer des répertoires à moins que vous ne soyez le super-utilisateur (root) et que l'option **-U** soit donnée à Perl. Même si ces conditions sont remplies, soyez conscient que l'effacement d'un répertoire par *unlink* peut endommager votre système de fichiers. C'est même carrément interdit sur certains systèmes de fichiers. Utilisez *rmdir()* à la place.

Si *LISTE* est omis, s'applique à *\$\_*.

**unpack** *TEMPLATE,EXPR*

*unpack()* réalise l'opération inverse de *pack()* : il prend une chaîne représentant une structure, la décompose en une liste de valeurs et retourne la liste de ces valeurs. (Dans un contexte scalaire, il retourne simplement la première valeur produite.)



La chaîne est découpée en morceaux selon le format TEMPLATE fourni. Chaque morceau est converti séparément en une valeur. Typiquement, soit la chaîne est le résultat d'un pack soit les octets de la chaîne représentent une structure C d'un certain type.

Le TEMPLATE a le même format que pour la fonction pack(). Voici une subroutine qui extrait une sous-chaîne :

```
sub substr {
 my($what,$where,$howmuch) = @_;
 unpack("x$where a$howmuch", $what);
}
```

et un autre exemple :

```
sub ordinal { unpack("c",$_[0]); } # identique à ord()
```

En plus, vous pouvez préfixer un champ avec un %<nombre> pour indiquer que vous voulez un checksum des items sur <nombre> bits à la place des items eux-mêmes. Par défaut, c'est un checksum sur 16 bits. Le checksum est calculé en additionnant les valeurs numériques des valeurs extraites (pour les champs alphanumériques, c'est la somme des ord(\$caractere) qui est prise et pour les champs de bits, c'est la somme des 1 et des 0).

Par exemple, le code suivant calcule le même nombre que le programme sum System V :

```
$checksum = do {
 local $/; # slurp!
 unpack("%32C*",<>) % 65535;
};
```

Le code suivant calcule de manière efficace le nombre de bits à un dans un vecteur de bits :

```
$setbits = unpack("%32b*", $selectmask);
```

Les formats p et P doivent être utilisés avec précautions. Puisque Perl n'a aucun moyen de vérifier que les valeurs passées à unpack() correspondent à des emplacements mémoires valides, le passage d'un pointeur dont on n'est pas sûr de la validité peut avoir des conséquences désastreuses.

Si il y a trop de champs ou si la valeur de répétition d'un champ ou d'un groupe est plus grande que ce que le reste de la chaîne d'entrée autorise, le résultat n'est pas vraiment défini : dans certains cas la valeur de répétition est diminuée ou alors unpack() produira des chaînes vide ou des zéros ou même se terminera par une erreur . Si la chaîne d'entrée est plus longue que ce qui est décrit par TEMPLATE, le reste est ignoré.

Voir pack pour plus d'exemples et de remarques.

#### untie VARIABLE

Casse le lien entre une variable et un package. (Voir tie()). N'a aucun effet si la variable n'est liée à aucun package.

#### unshift TABLEAU,LISTE

Fait le contraire de shift(). Ou le contraire de push(), selon le point de vue. Ajoute la liste LISTE au début du tableau TABLEAU et retourne le nouveau nombre d'éléments du tableau.

```
unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Remarquez que LISTE est ajoutée d'un seul coup et non élément par élément. Donc les éléments restent dans le même ordre. Utilisez reverse() pour faire le contraire.

#### use Module VERSION LISTE

#### use Module VERSION

#### use Module LISTE

#### use Module

#### use VERSION

Importe dans le package courant quelques "sémantiques" du module spécifié, généralement en les attachant à certains noms de subroutines ou de variables de votre package. C'est exactement équivalent à :

```
BEGIN { require Module; import Module LISTE; }
```

sauf que Module *doit* être un mot (bareword).

VERSION peut être soit une valeur numérique telle que 5.006 qui sera alors comparée à \$], soit une valeur littérale de la forme v5.6.1 qui sera alors comparée à \$^V (c.-à-d. \$PERL\_VERSION). Si VERSION est plus grand que la version courante de l'interpréteur Perl alors il se produira une erreur fatale ; Perl n'essaiera même pas de lire le reste du fichier. À comparer avec "require" qui fait la même vérification mais lors de l'exécution.

L'utilisation d'une VERSION sous forme littérale (ex: v5.6.1) devrait être évitée puisque cela entraîne la production de messages d'erreurs erronés par les vieilles versions de Perl qui ne reconnaissent pas cette syntaxe. L'équivalent numérique est préférable.

```

use v5.6.1; # vérification de version à la compilation
use 5.6.1; # idem
use 5.005_03; # numéro de version numérique
 # (préférable pour des raisons de compatibilité)

```

C'est pratique si vous devez vérifier la version courante de Perl avant d'utiliser (via `use`) des modules qui ont changé de manière incompatible depuis les anciennes versions. (Nous essayons de le faire le moins souvent possible).

Le `BEGIN` force le `require` et le `import` lors de la compilation. Le `require` assure que le module est chargé en mémoire si il ne l'a pas déjà été. `import` n'est pas une fonction interne – c'est juste un appel à une méthode statique ordinaire dans le package "Module" pour lui demander d'importer dans le package courant la liste de ses fonctionnalités. Le module peut implémenter sa méthode `import()` comme il le veut, bien que la plupart des modules préfèrent simplement la définir par héritage de la classe `Exporter` qui est définie dans le module `Exporter`. Voir *Exporter*. Si aucune méthode `import()` ne peut être trouvée alors l'appel est ignoré.

Si vous voulez éviter l'appel à la méthode "import" d'un package (par exemple pour éviter que votre espace de noms soit modifié), fournissez explicitement une liste vide :

```
use Module ();
```

C'est exactement équivalent à :

```
BEGIN { require Module }
```

Si l'argument `VERSION` est présent entre `Module` et `LISTE` alors `use` appelle la méthode `VERSION` de la classe `Module` avec la version spécifiée comme argument. La méthode `VERSION` par défaut, héritée de la classe `Universal`, crie (via `croak`) si la version demandée est plus grande que celle fournie par la variable `$Module::VERSION`.

À nouveau, il y a une différence entre omettre `LISTE` (`import` appelé sans argument) et fournir une `LISTE` explicitement vide `()` (`import` n'est pas appelé). Remarquez qu'il n'y a pas de virgule après `VERSION` !

Puisque c'est une interface largement ouverte, les pragmas (les directives du compilateur) sont aussi implémentés en utilisant ce moyen. Les pragmas actuellement implémentés sont :

```

use constant;
use diagnostics;
use integer;
use sigtrap qw(SEGV BUS);
use strict qw(subs vars refs);
use subs qw(afunc blurfl);
use warnings qw(all);
use sort qw(stable _quicksort _mergesort);

```

Certains d'entre eux sont des pseudo-modules qui importent de nouvelles sémantiques uniquement dans la portée du bloc courant (comme `strict` or `integer`) contrairement aux modules ordinaires qui importent des symboles dans le package courant (qui sont donc effectifs jusqu'à la fin du fichier lui-même).

Il y a une commande "no" permettant de "désimporter" des choses importées par `use`, i.e. elle appelle `unimport Module LISTE` à la place de `import`.

```

no integer;
no strict 'refs';
no warnings;

```

Voir *perlmodlib* pour une liste des modules et pragmas standard. Voir *perlrun* pour les options `-M` et `-m` qui permettent d'utiliser la fonctionnalité `use` depuis la ligne de commande.

### utime LISTE

Change la date d'accès et de modification de chaque fichier d'une liste de fichiers. Les deux premiers éléments de la liste doivent être, dans l'ordre, les dates NUMÉRIQUES d'accès et de modification. Retourne le nombre de fichiers modifiés avec succès. La date de modification de l'inode de chaque fichier est mis à l'heure courante. Par exemple, le code suivant a le même effet que la commande Unix `touch(1)` si les fichiers existent déjà et appartiennent à l'utilisateur qui exécute le programme :

```

#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;

```

Depuis perl 5.7.2, si les deux premiers éléments de `LIST` valent `undef` alors la fonction `utime(2)` de la bibliothèque C est appelée avec un second argument nul. Sur la plupart des systèmes, cela positionnera les dates d'accès et de modification des fichiers à l'heure courante (comme dans l'exemple ci-dessus) et pourra même marcher sur les fichiers d'autres utilisateurs pour lesquels vous possédez les droits d'écriture.

```
utime undef, undef, @ARGV;
```

Via NFS, c'est l'heure du serveur NFS qui sera utilisée et non celui de la machine locale. Si la synchronisation des horloges ne fonctionnent pas, ces deux machines pourront afficher des heures différentes. La commande `Unix touch(1)` utilise généralement cette méthode plutôt que celle montrée dans le premier exemple.

Notez que si un seul des deux premiers arguments vaut `undef`, cela revient à passer la valeur zéro et n'a donc pas le même effet que celui décrit lors du passage de deux `undef`. Dans ce cas, vous aurez aussi droit à un messages d'avertissement pour utilisation d'une valeur indéfinie.

### values HASH

Retourne la liste de toutes les valeurs contenues dans la table de hachage HASH. (Dans un contexte scalaire, retourne le nombre de valeurs.)

Les valeurs sont retournées dans un ordre apparemment aléatoire. Cet ordre peut changer avec les nouvelles versions de Perl mais vous avez la garantie qu'il est identique à celui produit par les fonctions `keys()` ou `each()` appliquées à la même table de hachage (tant qu'elle n'est pas modifiée). Depuis la version 5.8.1 de Perl, pour des raisons de sécurité, cet ordre change même à chaque exécution de Perl (voir *Attaques par complexité algorithmique in perlsec*).

Notez que les valeurs ne sont pas copiées. Ce qui signifie que leur modification entraîne la modification du contenu de la table de hachage :

```
for (values %hash) { s/foo/bar/g } # modifie les valeurs de %hash
for (@hash{keys %hash}) { s/foo/bar/g } # idem
```

Voir aussi `keys`, `each` et `sort`.

### vec EXPR,OFFSET,BITS

Traite la chaîne contenue dans EXPR comme un vecteur de bits dont les éléments sont de la largeur de BITS et retourne un entier non signé contenant la valeur du champ de bits spécifié par OFFSET. BITS spécifie le nombre de bits qui est occupé par chaque entrée dans le vecteur de bits. Cela doit être une puissance de deux entre 1 et 32 (ou 64 si votre plateforme le supporte).

Si BITS vaut 8, les "éléments" coïncident avec les octets de la chaîne d'entrée.

Si BITS vaut 16 ou plus, les octets de la chaîne d'entrée sont groupés par morceau de taille BITS/8 puis chaque groupe est converti en un nombre comme le feraient `pack()` et `unpack()` avec les formats big-endian n/N. Voir `pack` pour plus de détails.

Si BITS vaut 4 ou moins, la chaîne est découpée en octets puis les bits de chaque octet sont découpés en 8/BITS groupes. Les bits d'un octet sont numérotés à la manière little-endian comme dans `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. Par exemple, le découpage d'un seul octet d'entrée `chr(0x36)` en deux groupes donnera la liste (`0x6`, `0x3`) ; son découpage en 4 groupes donnera (`0x2`, `0x1`, `0x3`, `0x0`).

`vec` peut aussi être affecté auquel cas les parenthèses sont nécessaires pour donner les bonnes priorités :

```
vec($image, $max_x * $x + $y, 8) = 3;
```

Si l'élément sélectionné est en dehors de la chaîne, la valeur retournée sera zéro. Si, au contraire, vous cherchez à écrire dans un élément en dehors de la chaîne, Perl agrandira suffisamment la chaîne au préalable en la complétant par des octets nuls. L'écriture avant le début de la chaîne (c.-à-d avec un OFFSET négatif) est une erreur.

La chaîne ne devrait pas contenir de caractères ayant une valeur > 255 (ceci ne peut arriver que si vous utilisez l'encodage UTF-8). Quoiqu'il en soit, elle sera traitée sans tenir compte de son encodage UTF-8. Après une affectation via `vec`, les autres parties de votre programme ne considéreront plus cette chaîne comme étant encodée en UTF-8. En d'autres termes, si vous avez de tels caractères dans votre chaîne, `vec()` considérera les octets de la chaîne et non les caractères conceptuels qui la composent.

Les chaînes créées par `vec` peuvent aussi être manipulées par les opérateurs logiques `|`, `&` et `^` qui supposent qu'une opération bit à bit est voulue lorsque leurs deux opérandes sont des chaînes. Voir *Opérateurs bit à bit sur les chaînes in perlop*.

Le code suivant construit une chaîne ASCII disant 'PerlPerlPerl'. Les commentaires montrent la chaîne après chaque pas. Remarquez que ce code fonctionne de la même manière sur des machines big-endian ou little-endian.

```
my $foo = '';
vec($foo, 0, 32) = 0x5065726C; # 'Perl'

$foo eq "Perl" eq "\x50\x65\x72\x6C", 32 bits
print vec($foo, 0, 8); # affiche 80 == 0x50 == ord('P')
```



```

vec($_,11, 1) = 1 == 2048 0000000000001000000000000000000000
vec($_,12, 1) = 1 == 4096 0000000000000100000000000000000000
vec($_,13, 1) = 1 == 8192 0000000000000010000000000000000000
vec($_,14, 1) = 1 == 16384 0000000000000001000000000000000000
vec($_,15, 1) = 1 == 32768 0000000000000000100000000000000000
vec($_,16, 1) = 1 == 65536 0000000000000000010000000000000000
vec($_,17, 1) = 1 == 131072 0000000000000000001000000000000000
vec($_,18, 1) = 1 == 262144 0000000000000000000100000000000000
vec($_,19, 1) = 1 == 524288 0000000000000000000010000000000000
vec($_,20, 1) = 1 == 1048576 0000000000000000000001000000000000
vec($_,21, 1) = 1 == 2097152 0000000000000000000000100000000000
vec($_,22, 1) = 1 == 4194304 0000000000000000000000010000000000
vec($_,23, 1) = 1 == 8388608 0000000000000000000000001000000000
vec($_,24, 1) = 1 == 16777216 0000000000000000000000000100000000
vec($_,25, 1) = 1 == 33554432 0000000000000000000000000010000000
vec($_,26, 1) = 1 == 67108864 0000000000000000000000000001000000
vec($_,27, 1) = 1 == 134217728 0000000000000000000000000000100000
vec($_,28, 1) = 1 == 268435456 0000000000000000000000000000010000
vec($_,29, 1) = 1 == 536870912 0000000000000000000000000000001000
vec($_,30, 1) = 1 == 1073741824 000000000000000000000000000000010
vec($_,31, 1) = 1 == 2147483648 000000000000000000000000000000001
vec($_, 0, 2) = 1 == 1 1000000000000000000000000000000000000000
vec($_, 1, 2) = 1 == 4 0010000000000000000000000000000000000000
vec($_, 2, 2) = 1 == 16 0000100000000000000000000000000000000000
vec($_, 3, 2) = 1 == 64 0000001000000000000000000000000000000000
vec($_, 4, 2) = 1 == 256 0000000010000000000000000000000000000000
vec($_, 5, 2) = 1 == 1024 0000000000100000000000000000000000000000
vec($_, 6, 2) = 1 == 4096 0000000000001000000000000000000000000000
vec($_, 7, 2) = 1 == 16384 0000000000000010000000000000000000000000
vec($_, 8, 2) = 1 == 65536 0000000000000000100000000000000000000000
vec($_, 9, 2) = 1 == 262144 0000000000000000001000000000000000000000
vec($_,10, 2) = 1 == 1048576 0000000000000000000010000000000000000000
vec($_,11, 2) = 1 == 4194304 0000000000000000000000100000000000000000
vec($_,12, 2) = 1 == 16777216 00000000000000000000000010000000000000000
vec($_,13, 2) = 1 == 67108864 000000000000000000000000001000000000000000
vec($_,14, 2) = 1 == 268435456 00000000000000000000000000001000000000000000
vec($_,15, 2) = 1 == 1073741824 0000000000000000000000000000001000000000000000
vec($_, 0, 2) = 2 == 2 0100
vec($_, 1, 2) = 2 == 8 000100
vec($_, 2, 2) = 2 == 32 00000100000000000000000000000000000000000000
vec($_, 3, 2) = 2 == 128 00000001000000000000000000000000000000000000
vec($_, 4, 2) = 2 == 512 00000000010000000000000000000000000000000000
vec($_, 5, 2) = 2 == 2048 00000000000100000000000000000000000000000000
vec($_, 6, 2) = 2 == 8192 00000000000001000000000000000000000000000000
vec($_, 7, 2) = 2 == 32768 00000000000000100000000000000000000000000000
vec($_, 8, 2) = 2 == 131072 00000000000000001000000000000000000000000000
vec($_, 9, 2) = 2 == 524288 00000000000000000010000000000000000000000000
vec($_,10, 2) = 2 == 2097152 00000000000000000000100000000000000000000000
vec($_,11, 2) = 2 == 8388608 00000000000000000000001000000000000000000000
vec($_,12, 2) = 2 == 33554432 00000000000000000000000010000000000000000000
vec($_,13, 2) = 2 == 134217728 00000000000000000000000000100000000000000000
vec($_,14, 2) = 2 == 536870912 00000000000000000000000000001000000000000000
vec($_,15, 2) = 2 == 2147483648 000000000000000000000000000000001000000000000001
vec($_, 0, 4) = 1 == 1 100
vec($_, 1, 4) = 1 == 16 00001000000000000000000000000000000000000000
vec($_, 2, 4) = 1 == 256 00000000100000000000000000000000000000000000
vec($_, 3, 4) = 1 == 4096 000000000001000000000000000000000000000000000
vec($_, 4, 4) = 1 == 65536 000000000000001000000000000000000000000000000
vec($_, 5, 4) = 1 == 1048576 000000000000000010000000000000000000000000000
vec($_, 6, 4) = 1 == 16777216 000000000000000000100000000000000000000000000
vec($_, 7, 4) = 1 == 268435456 000000000000000000001000000000000000000000000

```

```

vec($_, 0, 4) = 2 == 2 01000000000000000000000000000000
vec($_, 1, 4) = 2 == 32 00000100000000000000000000000000
vec($_, 2, 4) = 2 == 512 00000000001000000000000000000000
vec($_, 3, 4) = 2 == 8192 00000000000000100000000000000000
vec($_, 4, 4) = 2 == 131072 0000000000000000001000000000000000
vec($_, 5, 4) = 2 == 2097152 000000000000000000000000100000000000
vec($_, 6, 4) = 2 == 33554432 000000000000000000000000000010000000
vec($_, 7, 4) = 2 == 536870912 00000000000000000000000000000000100
vec($_, 0, 4) = 4 == 4 0010000000000000000000000000000000
vec($_, 1, 4) = 4 == 64 0000001000000000000000000000000000
vec($_, 2, 4) = 4 == 1024 0000000000010000000000000000000000
vec($_, 3, 4) = 4 == 16384 0000000000000000100000000000000000
vec($_, 4, 4) = 4 == 262144 0000000000000000000010000000000000
vec($_, 5, 4) = 4 == 4194304 0000000000000000000000001000000000
vec($_, 6, 4) = 4 == 67108864 0000000000000000000000000000100000
vec($_, 7, 4) = 4 == 1073741824 0000000000000000000000000000000010
vec($_, 0, 4) = 8 == 8 0001000000000000000000000000000000
vec($_, 1, 4) = 8 == 128 0000000100000000000000000000000000
vec($_, 2, 4) = 8 == 2048 0000000000001000000000000000000000
vec($_, 3, 4) = 8 == 32768 0000000000000000001000000000000000
vec($_, 4, 4) = 8 == 524288 0000000000000000000000100000000000
vec($_, 5, 4) = 8 == 8388608 0000000000000000000000001000000000
vec($_, 6, 4) = 8 == 134217728 0000000000000000000000000000100000
vec($_, 7, 4) = 8 == 2147483648 0000000000000000000000000000000001
vec($_, 0, 8) = 1 == 1 1000000000000000000000000000000000
vec($_, 1, 8) = 1 == 256 0000000010000000000000000000000000
vec($_, 2, 8) = 1 == 65536 0000000000000000000100000000000000
vec($_, 3, 8) = 1 == 16777216 000000000000000000000000000010000000
vec($_, 0, 8) = 2 == 2 0100000000000000000000000000000000
vec($_, 1, 8) = 2 == 512 00000000010000000000000000000000
vec($_, 2, 8) = 2 == 131072 0000000000000000000010000000000000
vec($_, 3, 8) = 2 == 33554432 000000000000000000000000000010000000
vec($_, 0, 8) = 4 == 4 0010000000000000000000000000000000
vec($_, 1, 8) = 4 == 1024 00000000001000000000000000000000
vec($_, 2, 8) = 4 == 262144 0000000000000000000010000000000000
vec($_, 3, 8) = 4 == 67108864 0000000000000000000000000000100000
vec($_, 0, 8) = 8 == 8 0001000000000000000000000000000000
vec($_, 1, 8) = 8 == 2048 00000000000100000000000000000000
vec($_, 2, 8) = 8 == 524288 00000000000000000000100000000000
vec($_, 3, 8) = 8 == 134217728 0000000000000000000000000000100000
vec($_, 0, 8) = 16 == 16 0000100000000000000000000000000000
vec($_, 1, 8) = 16 == 4096 00000000000010000000000000000000
vec($_, 2, 8) = 16 == 1048576 0000000000000000000000100000000000
vec($_, 3, 8) = 16 == 268435456 00000000000000000000000000001000
vec($_, 0, 8) = 32 == 32 0000010000000000000000000000000000
vec($_, 1, 8) = 32 == 8192 00000000000001000000000000000000
vec($_, 2, 8) = 32 == 2097152 0000000000000000000000001000000000
vec($_, 3, 8) = 32 == 536870912 000000000000000000000000000000100
vec($_, 0, 8) = 64 == 64 0000001000000000000000000000000000
vec($_, 1, 8) = 64 == 16384 00000000000000100000000000000000
vec($_, 2, 8) = 64 == 4194304 0000000000000000000000001000000000
vec($_, 3, 8) = 64 == 1073741824 0000000000000000000000000000000010
vec($_, 0, 8) = 128 == 128 00000001000000000000000000000000
vec($_, 1, 8) = 128 == 32768 00000000000000010000000000000000
vec($_, 2, 8) = 128 == 8388608 0000000000000000000000001000000000
vec($_, 3, 8) = 128 == 2147483648 00000000000000000000000000000001

```

**wait**

Se comporte comme l'appel système `wait(2)` sur votre système : attend qu'un processus fils se termine et retourne le pid de ce processus ou -1 s'il n'y a pas de processus fils. Le statut est retourné par `$?`. Remarquez qu'une valeur de retour -1 peut signifier que les processus fils ont été automatiquement collectés tel que décrit dans *perlipc*.

**waitpid PID,FLAGS**

Attend qu'un processus fils particulier se termine et retourne le pid de ce processus ou -1 si ce processus n'existe pas. Sur certains système, une valeur 0 signifie qu'il y a encore des processus actifs. Le statut est retourné par \$?. Si vous dites :

```
use POSIX ":sys_wait_h";
#...
do {
 $kid = waitpid(-1,&WNOHANG);
} until $kid == -1;
```

alors vous pouvez réaliser une attente non bloquante sur plusieurs processus. Les attentes non bloquantes sont disponibles sur les machines qui connaissent l'un des deux appels système waitpid(2) ou wait4(2). Par contre, l'attente d'un process particulier avec FLAGS à 0 est implémenté partout. (Perl émule l'appel système en se souvenant des valeurs du statut des processus qui ont terminé mais qui n'ont pas encore été collectées par le script Perl.)

Remarquez que sur certains systèmes, une valeur de retour -1 peut signifier que les processus fils ont été automatiquement collectés. Voir *perlipc* pour les détails et d'autres exemples.

**wantarray**

Retourne true (vrai) si le contexte d'appel de la subroutine ou du eval en cours d'exécution attend une liste de valeurs. Retourne false (faux) si le contexte attend un scalaire. Retourne la valeur undef si le contexte n'attend aucune valeur ("void context" ou contexte vide).

```
return unless defined wantarray; # inutile d'en faire plus
my @a = complex_calculation();
return wantarray ? @a : "@a";
```

Le résultat de wantarray n'est pas défini au niveau le plus haut d'un fichier, dans un bloc BEGIN, CHECK, INIT ou END ou dans une méthode DESTROY.

Cette fonction aurait dû s'appeler wantlist().

**warn LISTE**

Produit un message d'erreur sur STDERR exactement comme die() mais ne quitte pas et ne génère pas d'exception.

Si LISTE est vide et si \$@ contient encore une valeur (provenant par exemple d'un eval précédent) alors cette valeur est utilisée après y avoir ajouté "\t...caught". C'est pratique pour s'approcher d'un comportement presque similaire à celui de die().

Si \$@ est vide alors la chaîne "Warning: Something's wrong" (N.d.t : "Attention: quelque chose va mal") est utilisée.

Aucun message n'est affiché si une subroutine est attachée à \$SIG{\_\_WARN\_\_}. C'est de la responsabilité de cette subroutine de gérer le message comme elle le veut (en le convertissant en un die() par exemple). La plupart des routines du genre devraient s'arranger pour afficher réellement les messages qu'elles ne sont pas prêtes à recevoir en appelant à nouveau warn(). Remarquez que cela fonctionne sans produire une boucle sans fin puisque les routines attachées à \_\_WARN\_\_ ne sont pas appelés à partir d'une subroutine attachée.

Ce comportement est complètement différent de celui des routine attachées à \$SIG{\_\_DIE\_\_} (qui ne peuvent pas supprimer le texte d'erreur mais seulement le remplacer en appelant à nouveau die()).

L'utilisation d'une subroutine attachée à \_\_WARN\_\_ fournit un moyen puissant pour supprimer tous les messages d'avertissement (même ceux considérés comme obligatoires). Un exemple :

```
supprime *tous* les messages d'avertissement lors de la compilation
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20; # pas d'avertissement pour la duplication de
 # $foo... mais c'est ce qu'on voulait !
pas de messages d'avertissement avant ici
$DOWARN = 1;

messages d'avertissement à partir d'ici
warn "\$foo is alive and $foo!"; # devrait apparaître
```

Voir *perlvar* pour plus de détails sur la modification des entrées de %SIG et pour plus d'exemples. Voir le module Carp pour d'autres sortes d'avertissement utilisant les fonctions carp() et cluck().

**write DESCRIPTEUR**

**write EXPR****write**

Écrit un enregistrement formaté (éventuellement multi-lignes) vers le DESCRIPTEUR spécifié en utilisant le format associé à ce fichier. Par défaut, le format pour un fichier est celui qui a le même nom que le descripteur mais le format du canal de sortie courant (voir la fonction `select()`) peut être spécifié explicitement en stockant le nom du format dans la variable `$^`.

Le calcul de l'en-tête est fait automatiquement : si il n'y a pas assez de place sur la page courante pour l'enregistrement formaté, on passe à la page suivante en affichant un format d'en-tête spécial puis on y écrit l'enregistrement formaté. Par défaut, le nom du format d'en-tête spécial est le nom du descripteur auquel on ajoute "\_TOP" mais il peut être dynamiquement modifié en affectant le nom du format voulu à la variable `$^` lorsque le descripteur est sélectionné (par `select()`). Le nombre de lignes restant dans la page courante est donné par la variable `$-` qui peut être mise à 0 pour forcer le passage à la page suivante.

Si DESCRIPTEUR n'est pas spécifié, le sortie se fait sur le canal de sortie courant qui, au début, est `STDOUT` mais qui peut être changé par l'opérateur `select()`. Si le DESCRIPTEUR est une expression EXPR alors l'expression est évaluée et la chaîne résultante est utilisée comme nom du descripteur à utiliser. Pour en savoir plus sur les formats, voir *perldata*.

Notez que `write` n'est PAS le contraire de `read()`. Malheureusement.

**y///**

L'opérateur de translittération. Identique à `tr///`. Voir *perlop*.

## 31.2 TRADUCTION

### 31.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 31.2.2 Traducteur

Traduction initiale : Paul Gaborit ([paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)), Jean-Pascal Peltier ([jp\\_peltier@altavista.net](mailto:jp_peltier@altavista.net)).

Mise à jour en 5.6.0, en 5.8.0 et en 5.8.8 : Paul Gaborit ([paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)).

### 31.2.3 Relecture

Gérard Delafond. Jean-Louis Morel.



# Chapitre 32

## perlopentut

Apprenez à ouvrir (des fichiers) à la mode Perl

### 32.1 DESCRIPTION

En Perl il existe deux manières standard d'ouvrir des fichiers : la syntaxe à la shell intéressante pour son côté pratique et la syntaxe C pour sa précision. La syntaxe shell se décline elle-même en formes à 2- et 3-paramètres dont la sémantique diffère. Vous avez le choix.

### 32.2 Open à la shell

La fonction `open` de Perl a été conçue pour imiter le fonctionnement des opérateurs de redirection du shell. Voici quelques exemples simples en shell :

```
$ monprogramme fichier1 fichier2 fichier3
$ monprogramme < fichiersource
$ monprogramme > fichiercible
$ monprogramme >> fichiercible
$ monprogramme | autreprogramme
$ autreprogramme | monprogramme
```

Et voici quelques exemples plus complexes :

```
$ autreprogramme | monprogramme f1 - f2
$ autreprogramme 2>&1 | monprogramme -
$ monprogramme <&3
$ monprogramme >&4
```

Les programmeurs habitués aux formes précédentes apprécieront d'apprendre que Perl supporte directement ces constructions familières en utilisant une syntaxe quasiment identique à celle du shell.

#### 32.2.1 Opens simples

La fonction `open` prend deux arguments : le premier est un manipulateur de fichier (appelé aussi handle de fichier ou filehandle) et le second est une simple chaîne de caractères indiquant à la fois quel fichier ouvrir et comment l'ouvrir. `open` retourne vrai quand tout s'est bien passé, et faux en cas d'échec en positionnant la variable interne `$!` pour indiquer quelle erreur système a provoqué l'échec. Si le handle du fichier était ouvert avant l'appel, il est implicitement fermé avant réouverture.

Quelques exemples :

```

open(INFO,"fichierdedonnees")
 || die("l'ouverture du fichierdedonnees a échoué: $!");
open(INFO,"< fichierdedonnees")
 || die("l'ouverture du fichierdedonnees a échoué: $!");
open(RES, "> fichierstats")
 || die("l'ouverture du fichierstats a échoué: $!");
open(LOG, ">> fichierlog ")
 || die("l'ouverture du fichierlog a échoué: $!");

```

Si vous préférez économiser la ponctuation, vous pouvez aussi l'écrire ainsi :

```

open INFO,"< fichierdedonnees"
 or die "l'ouverture du fichierdedonnees a échoué: $!";
open RES, "> fichierstats"
 or die "l'ouverture du fichierstats a échoué: $!";
open LOG, ">> fichierlog "
 or die "l'ouverture du fichierlog a échoué: $!";

```

Il y a quelques détails à remarquer. D'abord, le symbole inférieur initial est optionnel. S'il est omis, Perl suppose que vous voulez ouvrir le fichier en lecture.

Remarquez aussi que le premier exemple utilise l'opérateur logique `||`, tandis que le second utilise `or`, dont la précedence est plus faible. L'utilisation de `||` dans la seconde série d'exemples signifie en fait :

```

open INFO, ("< fichierdedonnees" || die "l'ouverture a échoué: $!");

```

qui n'est certainement pas ce que vous souhaitez.

L'autre point important à noter est que, comme sous un shell Unix, les espaces qui suivent ou précèdent le nom de fichier sont ignorés. C'est une bonne chose car vous ne voudriez pas que les commandes qui suivent aient des comportements différents :

```

open INFO, "<fichier"
open INFO, "< fichier"
open INFO, "< fichier"

```

Ignorer les espaces entourant le nom aide également quand vous lisez ce nom depuis un autre fichier et oubliez de nettoyer les blancs indésirables avant l'ouverture :

```

$filename = <INFO>; # houps, \n est encore là
open(EXTRA, "< $fichier") || die "l'ouverture de $filename a échoué: $!";

```

Ce n'est pas un bug, c'est volontaire. `open` imite le shell dans son utilisation des caractères de redirection pour spécifier le type d'ouverture désiré, il se comporte également comme le shell dans le traitement des espaces surnuméraires. Pour accéder à des fichiers portant des noms ennuyeux, allez lire Exorciser la magie (§32.4.2).

Il existe aussi une version à trois paramètres de `open`, qui vous laisse isoler le caractère de redirection dans son propre champ :

```

open(INFO, ">", $fichier) || die "Impossible de créer $fichier: $!";

```

Dans ce cas le nom du fichier à ouvrir est précisément la chaîne de caractères contenue dans `$fichier`, vous n'avez donc plus à vous soucier de savoir si `$fichier` contient des caractères susceptibles d'influencer le mode d'ouverture, ou des espaces au début du nom de fichier qui seraient absorbés dans un `open` à deux paramètres. Diminuer les interpolations inutiles de chaînes est aussi une bonne chose.

### 32.2.2 Handles de fichier indirects

Le premier paramètre de `open` peut être une référence vers un handle de fichier. Depuis perl 5.6.0, si ce paramètre n'est pas initialisé, Perl crée automatiquement un handle de fichier et place une référence vers ce handle dans le premier paramètre, comme ça :

```
open(my $in, $infile) or die "Pas moyen d'ouvrir $infile: $!";
while (<$in>) {
 # utiliser $_
}
close $in;
```

Les handles de fichiers indirects facilitent la gestion des espaces de noms. Puisque les handles de fichiers sont globaux au paquetage courant, deux sous-routines qui essaieraient d'ouvrir `INFILE` entreraient en conflit. Avec deux fonctions ouvrant des handles de fichier indirects du genre `my $infile`, il n'y a plus de risque et il est inutile de s'inquiéter de futurs conflits hypothétiques.

Un autre comportement bien pratique est qu'un handle de fichier indirect est automatiquement fermé quand il sort de la portée courante ou quand on affecte `undef` à la variable :

```
sub premiereligne {
 open(my $in, shift) && return scalar <$in>;
 # pas de close() nécessaire
}
```

### 32.2.3 L'ouverture des tubes

En C, quand vous voulez ouvrir un fichier en utilisant la librairie d'E/S standard vous utilisez la fonction `fopen`. Et pour ouvrir un tube vous utilisez la fonction `popen`. Par contre en shell vous utilisez simplement un caractère de redirection différent. C'est aussi la méthode retenue en Perl. L'appel à `open` reste le même ; c'est juste son paramètre qui diffère.

Si le premier caractère du second paramètre d'`open` est un symbole tube (une barre verticale), `open` lance une nouvelle commande et ouvre un handle de fichier en écriture seule qui transmet des données à cette commande. Cela vous permet d'écrire dans ce handle afin d'envoyer ce que vous voulez à l'entrée standard de la commande. Exemple :

```
open(IMPRIMANTE, "| lpr -Plp1") || die "Erreur d'exécution de lpr: $!";
print IMPRIMANTE "machin\n";
close(IMPRIMANTE) || die "Erreur de fermeture de lpr: $!";
```

Si le dernier caractère du second paramètre d'`open` est un tube, vous lancez une nouvelle commande en ouvrant un handle de fichier en lecture seule associé à la sortie de cette commande. Via ce handle de fichier, vous pouvez ainsi lire tout ce que la commande produit sur sa sortie standard. Exemple :

```
open(RESEAU, "netstat -i -n |") || die "Erreur d'exécution de netstat: $!";
while (<RESEAU>) { } # utiliser les données reçues
close(RESEAU) || die "Erreur de fermeture de netstat: $!";
```

Que se passe-t-il si vous essayez d'ouvrir un tube depuis ou à partir d'une commande inexistante ? Dans la mesure du possible, Perl essaye de détecter l'échec et de positionner `#!` comme d'habitude. Toutefois, si la commande contient certains caractères spéciaux tels que `>` or `*` (les 'metacaractères'), Perl n'exécute pas directement la commande. À la place, Perl lance un shell, qui lui-même essaye d'exécuter la commande. Cela signifie que c'est le shell qui reçoit le code d'erreur. Dans un tel cas, l'appel à `open` indiquera uniquement un échec si Perl ne peut pas exécuter un shell. Voyez Comment capturer `STDERR` à partir d'une commande externe ? in *perlfaq8* pour voir comment traiter ce cas. Vous trouverez une seconde explication dans *perlipc*.

Si vous voulez ouvrir un tube bidirectionnel, la bibliothèque `IPC::Open2` s'en chargera pour vous. Consultez *Communications bidirectionnelles avec un autre processus* in *perlipc*.

### 32.2.4 Le fichier moins ou tirt

Suivant, encore une fois, l'exemple des utilitaires standard du shell Unix, la fonction `open` traite de manière particulière un fichier dont le nom est un tirt (le signe moins "-"). Ouvrir tirt en lecture signifie en réalité accéder à l'entrée standard. Ouvrir tirt en écriture signifie accéder à la sortie standard.

Si le tirt peut être utilisé en tant qu'entrée ou sortie par défaut, que va-t-il se passer si on ouvre un tube vers ou à partir d'un tirt ? Quelle commande par défaut sera lancée ? Réponse : le script même que vous êtes en train d'exécuter ! Il s'agit en réalité d'un `fork` dissimulé à l'intérieur d'un appel à `open`. Référez-vous à Ouvrir des tubes en toute sécurité in *perlpc* pour les détails.

### 32.2.5 Mélanger la lecture et l'écriture

Il est possible d'ouvrir un fichier à la fois en lecture et en écriture. Il suffit d'ajouter un "+" devant le symbole de redirection. Comme en shell l'utilisation du symbole "inférieur" ne crée jamais de nouveau fichier et ne peut ouvrir qu'un fichier pré-existant. À l'inverse utiliser un symbole "supérieur" détruit (tronque à longueur zéro) un éventuel fichier pré-existant ou crée un nouveau fichier s'il n'en existe pas encore de ce nom. L'ajout d'un "+" pour la lecture/écriture n'affecte pas le fonctionnement de la fonction il tronque toujours l'existant ou crée un nouveau fichier.

```
open(WTMP, "+< /usr/adm/wtmp")
 || die "Impossible d'ouvrir /usr/adm/wtmp: $!";

open(SCREEN, "+> lkscreen")
 || die "Impossible d'ouvrir lkscreen: $!";

open(LOGFILE, "+>> /var/log/applog")
 || die "Impossible d'ouvrir /var/log/applog: $!";
```

Le premier exemple ne crée jamais de nouveau fichier et le second détruit toujours un fichier existant. Le troisième exemple crée un nouveau fichier si nécessaire mais sans effacer un fichier existant, et permet de lire à n'importe quel point du fichier, mais toutes les écritures auront lieu en fin de fichier. En bref, le premier cas est assez banal, bien plus que le second et le troisième, qui sont presque toujours des erreurs. (Si vous connaissez le C, le signe plus dans le `open` de Perl dérive historiquement de celui du `fopen(3S)` du C, fonction qui est appelée au bout du compte).

En pratique, si vous avez besoin de mettre à jour un fichier, à moins de travailler sur un fichier binaire comme dans le cas de `WTMP` ci-dessus, vous aurez tout intérêt à utiliser une autre approche. Le bon choix c'est l'option ligne de commande `-i` de Perl. La commande suivante prend un ensemble de sources C, C++, yacc et fichiers d'en-tête et remplace tous les `foo` par `bar` dans le corps de ces fichiers, tout en laissant intacts les fichiers d'origine avec un ".orig" à la fin de leur noms :

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *. [Cchy]
```

C'est la manière courte de pratiquer un petit manège de renommage de fichiers fréquemment employé par les développeurs qui est en pratique la meilleure façon de mettre à jour des fichiers texte. Elle minimise notamment les risques de pertes de données et facilite et accélère le traitement. Voyez la question numéro 2 dans *perlfaq5* pour plus de détails.

### 32.2.6 Les filtres

L'une des utilisations les plus fréquentes de `open` est si discrète que vous ne l'avez probablement jamais remarquée. Quand vous traitez les handle de fichier de ARGV en utilisant `<ARGV>`, Perl effectue en réalité un `open` implicite de chaque fichier de `@ARGV`. Ainsi un programme appelé par :

```
$ monprogramme fichier1 fichier2 fichier3
```

parviendra à ouvrir et traiter un par un tous les fichiers passés en ligne de commande en utilisant une construction très simple du genre :

```
while (<>) {
 # utiliser $_
}
```

Si `@ARGV` est vide au premier tour de boucle, Perl fait comme si vous aviez ouvert le fichier moins, c'est-à-dire l'entrée standard. En fait, `$ARGV`, le fichier en cours d'utilisation lors du traitement de `<ARGV>`, est même fixé à `"-"` dans de telles circonstances.

Vous êtes invité à pré-traiter le contenu d'`@ARGV` avant d'entamer la boucle pour vous assurer que son contenu vous convient. Une raison de procéder ainsi peut être par exemple de supprimer d'`@ARGV` les options de ligne de commande débutant par un signe moins. Même si rien ne vous empêche de gérer les cas simple à la main, simplifiez vous la vie avec la famille de modules `getopts` :

```
use Getopt::Std;

-v, -D, -o ARG, positionnent $opt_v, $opt_D, $opt_o
getopts("vDo:");

-v, -D, -o ARG, positionnent $args{v}, $args{D}, $args{o}
getopts("vDo:", \%args);
```

Ou le module standard `Getopt::Long` qui autorise les noms de paramètres longs :

```
use Getopt::Long;
GetOptions("verbose" => \$verbose, # --verbose
 "Debug" => \$debug, # --Debug
 "output=s" => \$output);
--output=quelquechose ou --output quelquechose
```

Une autre raison de préparer la liste d'arguments peut être de choisir comme convention qu'une liste d'arguments vide signifie "tous les fichiers du répertoire courant";

```
@ARGV = glob("*") unless @ARGV;
```

Vous pouvez même choisir de conserver exclusivement les paramètres correspondant à des fichiers texte normaux. La technique ci-dessous est un peu laconique et vous pouvez bien entendu choisir d'afficher au passage les noms des fichiers retenus :

```
@ARGV = grep { -f && -T } @ARGV;
```

Si vous utilisez les options en ligne de commande `-n` ou `-p` de Perl, les modifications à `@ARGV` devront être placées dans un bloc `BEGIN{}`.

Souvenez-vous qu'un `open` normal n'a aucune propriété particulière, dans la mesure où il pourrait aussi bien appeler `fo-pen(3S)` ou `popen(3S)`, en fonction de ses arguments ; c'est la raison pour laquelle on l'appelle parfois le "open magique". Voici un exemple :

```
$pwdinfo = 'domainname' =~ /\^(\\(none\\))?$/
 ? '< /etc/passwd'
 : 'ypcat passwd |';

open(PWD, $pwdinfo)
 or die "Impossible d'ouvrir $pwdinfo: $!";
```

Ce type de construction est aussi utilisée pour l'écriture de filtres. Dans la mesure où le traitement de `<ARGV>` utilise le `open` normal de Perl (avec la syntaxe à la mode shell) il profite de toutes les particularités que nous avons pu voir. Exemple :

```
$ monprogramme f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

Ce programme lit des données depuis le fichier `f1`, the processus `cmd1`, l'entrée standard (`tmpfile` ici puisqu'elle est redirigée), le fichier `f2`, la commande `cmd2`, et enfin fichier `f3`.

Oui, cela signifie aussi que si vous avez des fichiers nommés `"-"` (ou du même accabit) dans votre répertoire, `open` ne les traitera pas comme des littéraux. Vous allez devoir passer des paramètres du genre `"./-"`, comme vous le feriez pour le programme `rm`, vous pouvez aussi choisir d'utiliser `sysopen` de la manière décrite ci dessous.

Une application très intéressante consiste à remplacer les fichiers portant certains noms par des tubes. Par exemple pour la décompression automatique via `gzip` des fichiers `zippés` ou `compressés` :

```
@ARGV = map { /\^(gz|Z)$/ ? "gzip -dc $_|" : $_ } @ARGV;
```

Ou encore, si vous avez installé le programme `GET` qui vient avec `LWP`, vous pouvez télécharger les URLs avant de les traiter.

```
@ARGV = map { m#^\w+://# ? "GET $_|" : $_ } @ARGV;
```

Ce n'est pas pour rien qu'on appelle ça la magie de `<ARGV>`. Drôlement chiadé, non ?

## 32.3 Ouverture à la C

Si vous désirez la convivialité du shell alors choisissez sans hésiter, l'open façon Perl. En revanche si vous avez besoin de la finesse de contrôle qu'offre le simplissime fopen(3S) du C vous devriez jeter un coup d'oeil à la fonction Perl sysopen, une simple interface vers l'appel système open(2). Cela signifie qu'il est légèrement plus tarabiscoté, mais c'est le prix de la précision.

sysopen prend 3 (ou 4) arguments.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

Le paramètre HANDLE est un handle de fichier, exactement comme pour open. Le PATH est littéralement un chemin d'accès, qui ne tient donc aucun compte des caractères supérieur, inférieur, tube ou moins qu'il peut contenir et qui n'ignore pas non plus les espaces. Si un tel caractère est là c'est qu'il fait partie du chemin d'accès. Le paramètre FLAGS contient une ou plusieurs valeurs issues du module Fcntl et combinés par l'opérateur ou bit-à-bit "|".

Le dernier paramètre, le MASK, est optionnel ; s'il est présent, il est combiné avec l'umask courant de l'utilisateur pour générer le mode de création du fichier. Vous pouvez généralement l'omettre.

Bien que traditionnellement la lecture seule, l'écriture seule ou la lecture écriture soient respectivement associées aux valeurs 0, 1, et 2, ce n'est pas le cas sur certains systèmes d'exploitation. Il est donc préférable de charger le module Fcntl, qui fournit les constantes symboliques standard suivantes pour le paramètre FLAGS :

O_RDONLY	Lecture seule
O_WRONLY	Écriture seule
O_RDWR	Lecture et Écriture
O_CREAT	Créer le fichier s'il n'existe pas
O_EXCL	Échouer si le fichier existe
O_APPEND	Ajout à la fin du fichier
O_TRUNC	Tronquer le fichier
O_NONBLOCK	Accès non bloquant

Les constantes moins courantes suivantes sont disponibles ou non selon le système d'exploitation O\_BINARY, O\_TEXT, O\_SHLOCK, O\_EXLOCK, O\_DEFER, O\_SYNC, O\_ASYNC, O\_DSYNC, O\_RSYNC, O\_NOCTTY, O\_NDELAY and O\_LARGEFILE. Consultez la page de manuel open(2) ou son équivalent local pour les détails. (Note : à partir de Perl version 5.6 la constante O\_LARGEFILE, est automatiquement ajoutée à sysopen() si elle est disponible. En effet, les grands fichiers sont le défaut.)

Voici comment utiliser sysopen pour émuler certains des appels à open que nous avons montré. Nous omettrons la gestion d'erreur `|| die $!` pour ne pas compliquer les exemples, mais assurez vous de bien tester les valeurs de retour dans du code de production. Les exemples présentés ne sont pas exactement identiques aux précédents dans la mesure où open supprime les espaces en queue et en tête ce que ne fait pas sysopen, mais cela vous donnera l'idée générale.

Pour ouvrir un fichier en lecture :

```
open(FH, "< $path");
sysopen(FH, $path, O_RDONLY);
```

Pour ouvrir un fichier en écriture, créer un nouveau fichier si nécessaire ou tronquer un vieux fichier :

```
open(FH, "> $path");
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

Pour ouvrir un fichier en ajout, en le créant si nécessaire :

```
open(FH, ">> $path");
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

Pour ouvrir un fichier en mise à jour, le fichier devant déjà exister :

```
open(FH, "+< $path");
sysopen(FH, $path, O_RDWR);
```

Et voici ce que l'on peut faire avec `sysopen` qui est impossible avec le `open` standard. Comme vous pouvez le constater il ne s'agit que de contrôler l'indicateur figurant en troisième argument :

Pour ouvrir un fichier en écriture en créant un nouveau fichier, celui-ci ne devant pas préexister :

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

Pour ouvrir un fichier en ajout, ce fichier devant déjà exister :

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

Pour ouvrir un fichier en mise à jour, en créant un nouveau fichier si nécessaire :

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

Pour ouvrir un fichier en mise à jour, ce fichier ne devant pas préexister :

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

Pour ouvrir un fichier en mode non bloquant, en le créant si nécessaire :

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK | O_CREAT);
```

### 32.3.1 Permissions à la mode Unix

Si vous négligez le paramètre `MASK` de `sysopen`, Perl utilise la valeur octale 0666. Le zéro initial indique qu'il s'agit d'un nombre écrit en octal, le premier chiffre octal correspond aux droits du propriétaire du fichier, le second aux droits de son groupe et le troisième à ceux des autres. Le `MASK` normal pour les exécutable et les répertoires devrait être 0777, et pour tout le reste 0666.

Pourquoi une telle permissivité ? En fait cette façon de faire est loin d'être aussi permissive qu'il peut sembler. En effet comme nous l'avons dit le `MASK` est modifié par l'`umask` du processus actif. L'`umask` est un nombre représentant les bits de permission *désactivés*; c'est-à-dire les bits qui ne doivent pas être activés même s'ils sont explicitement demandés dans les permissions des fichiers créés.

Par exemple, si votre `umask` vaut 027, alors la partie 020 désactive les droits d'écriture pour le groupe, et la partie 007 empêche les autres de lire, écrire ou exécuter des fichiers. Dans un tel cas passer 0666 à `sysopen` crée un fichier avec le mode 0640, puisque `0666 & ~027` vaut 0640.

Vous n'aurez que très rarement besoin d'utiliser le paramètre `MASK` de `sysopen()`. En l'utilisant vous privez l'utilisateur de la liberté de choisir les droits des nouveaux fichiers. Forcer un choix est presque toujours mauvais. Une exception pourrait être le stockage de données sensibles ou privées, comme le mail, les fichiers de cookies et les fichiers temporaires internes.

## 32.4 Trucs obscurs avec open

### 32.4.1 Ré-ouverture de fichiers (dups)

De temps en temps on dispose déjà d'un handle vers un fichier ouvert et on veut obtenir un second handle identique au premier. Pour faire ce genre de choses en shell on place un et-commercial "&" devant le numéro du descripteur au niveau des redirections. Par exemple, `2>&1` prend le descripteur 2 (qui correspond à `STDERR` en Perl) et le redirige vers le descripteur 1 (qui correspond en général à `STDOUT`). Le principe est fondamentalement le même en Perl : un nom de fichier commençant par un et-commercial est traité comme un numéro de descripteur de fichier s'il s'agit d'un nombre ou comme un handle de fichier s'il s'agit d'une chaîne.

```
open(SAUVEOUT, ">&SAUVEERR") || die "Impossible de dupliquer SAUVEERR: $!";
open(MHCONTEXTTE, "<&4") || die "Impossible de dupliquer fd4: $!";
```

Cela signifie que si une fonction attend un nom de fichier mais que vous ne voulez ou ne pouvez pas lui indiquer un nom de fichier parce que le fichier en question est déjà ouvert, vous pouvez vous contenter du handle de fichier précédé d'un et-commercial. Il est tout de même préférable d'utiliser un handle pleinement qualifié, juste au cas où la fonction serait dans un autre paquetage :

```
unefunction("&main::FICHIERLOG");
```

De cette manière si unefunction() à l'intention d'ouvrir son paramètre il peut se contenter d'utiliser le handle déjà ouvert. Ce n'est pas la même chose que de passer un handle. En effet, dans le cas d'un handle vous n'avez pas la possibilité d'ouvrir le fichier. Ici vous avez l'opportunité de transmettre quelque chose à open.

Si vous disposez de l'un de ces complexes objets d'E/S C++, très à la mode dans certains milieux, alors cette technique ne fonctionne pas car ces objets ne sont pas des handle de fichier corrects dans le sens Perl. Vous allez devoir faire un appel à fileno() pour en extraire le véritable numéro de descripteur, si vous pouvez :

```
use IO::Socket;
$handle = IO::Socket::INET->new("www.perl.com:80");
$fd = $handle->fileno;
unefunction("&$fd"); # ce n'est pas un appel de fonction indirect
```

Il peut cependant être plus facile (et cela sera certainement plus rapide) d'utiliser simplement de vrais handles de fichiers :

```
use IO::Socket;
local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
die "Impossible de se connecter" unless defined(fileno(REMOTE));
unefunction("&main::REMOTE");
```

Si le handle de fichier ou le numéro de descripteur est précédé non pas d'un simple "&" mais plutôt d'une combinaison "&=", alors Perl ne crée pas tout à fait un nouveau descripteur ouvert sur le même fichier en utilisant l'appel système dup(2). Ce qui est créé est plutôt une sorte d'alias vers le descripteur existant en utilisant la bibliothèque système fdopen(3S). Cette technique est légèrement plus économique en ressources systèmes, même si c'est de moins en moins une préoccupation à notre époque. Voici ce que cela donne :

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fd") or die "Échec de l'appel à fdopen $fd: $!";
```

Si vous utilisez le <ARGV> magique, vous pourriez même utiliser cette syntaxe en tant qu'argument de ligne de commande de @ARGV, quelque chose du genre "<&=\$MHCONTEXTFD", mais personne ne l'a encore fait à notre connaissance.

### 32.4.2 Exorciser la magie

Perl est un langage plus intuitif que, disons Java. Intuitif signifie que les choses se passent généralement comme on le souhaite, même si l'on en comprend pas toujours le pourquoi du comment et que cela ressemble dans certains cas au lapin sortant du chapeau. Malheureusement cette approche conduit de temps à autres à ce qu'il se passe un peu trop de choses en coulisses. La magie de Perl tourne alors à la malédiction.

Si la version magique de open est un peu trop magique à votre gout, vous n'êtes pas obligé de vous résigner à utiliser sysopen. Pour ouvrir un fichier contenant des caractères spéciaux, il faut protéger les espaces en tête et en fin de nom. Les espaces en tête sont protégés en ajoutant simplement "./" en tête du nom de fichier débutant par des espaces. Les espaces en fin de nom sont protégés en ajoutant explicitement un octet ASCII NUL ("\0") en fin de chaîne.

```
$fichier =~ s#^\s#./$1#;
open(FH, "< $fichier\0") || die "Impossible d'ouvrir $fichier: $!";
```

Cette technique suppose, bien sûr, que sur votre système, le point désigne le dossier courant, le slash le séparateur de dossier et que les octets ASCII NUL sont interdits dans les noms de fichiers valides. La majorité des systèmes d'exploitation acceptent ces conventions, aussi bien les systèmes POSIX que les systèmes propriétaires Microsoft. L'unique système relativement répandu qui ne fonctionne pas ainsi est le système Macintosh "Classique", qui utilise les deux points (:) là où les autres ont opté pour le slash. Peut-être que sysopen n'est peut-être pas une si mauvaise idée après tout.

Si vous voulez traiter <ARGV> de façon mortellement ennuyeuse et non magique, vous pouvez commencer par faire ceci :

```
"Sam s'assit par terre et pris sa tête dans ses mains.
'Je souhaite ne jamais être venu ici et je ne veux plus jamais
voir de magie' dit-il, avant de se taire."
for (@ARGV) {
 s#^([\s./])#./$1#;
 $_ .= "\0";
}
while (<>) {
 # now process $_
}
```



Soyez conscient que les utilisateurs ne vont pas apprécier d'être privé du "-" pour désigner l'entrée standard comme ils en ont l'habitude.

### 32.4.3 Chemins d'accès comme opens

Vous avez sans doute remarqué le format de certains messages produits par les fonctions `warn` et `die` de Perl :

```
Un avertissement at nom_du_script line 29, <FH> line 7.
```

Le message signifie que vous avez ouvert un handle de fichier `FH` et lus sept enregistrements à partir de `FH`. Mais quel était donc le nom du fichier associé à ce handle ?

Si vous n'avez pas activé l'option `strict refs`, ou si vous l'avez temporairement désactivée, alors il vous suffit d'écrire :

```
open($chemin, "< $chemin") || die "Impossible d'ouvrir $chemin: $!";
while (<$chemin>) {
 # du code
}
```

Puisque vous utilisez le chemin d'accès au fichier en tant que handle, vos avertissements ressembleront à :

```
Un avertissement at nom du script line 29, </etc/motd> line 7.
```

### 32.4.4 Open à un seul argument

Vous vous souvenez que nous avons dit que le `open` de Perl prenait deux arguments. En réalité nous avons fait preuve d'une certaine mauvaise foi. C'est un mensonge par omission. En réalité `open` peut aussi ne prendre qu'un seul argument. À la condition exclusive d'utiliser une variable globale pour le handle de fichier, avec un lexical ça ne marche pas, vous pouvez ne passer à `open` qu'un seul paramètre, le handle de fichier, le nom de fichier sera extrait de la variable scalaire globale de même nom.

```
$FICHIER = "/etc/motd";
open FICHIER or die "Impossible d'ouvrir $FILE: $!";
while (<FICHIER>) {
 # du code
}
```

La raison de cette étrangeté ? Perl aussi comporte son lot d'horreurs hyst[é]riques. Ce comportement existe dans Perl quasiment depuis sa création, voire même avant.

### 32.4.5 Jouer avec STDIN et STDOUT

Une bonne idée avec `STDOUT` c'est de le fermer explicitement lorsque votre programme n'en a plus besoin.

```
END { close(STDOUT) || die "Impossible de fermer stdout: $!" }
```

Si vous ne le faites pas et que votre programme remplit la partition courante à cause d'une redirection de ligne de commande, l'erreur n'apparaîtra pas et le programme ne renverra pas de statut d'échec en sortie.

Vous n'êtes pas obligé d'accepter `STDIN` et le `STDOUT` tels qu'on vous les donne. Rien ne vous empêche de les réouvrir si vous le souhaitez :

```
open(STDIN, "< fichier")
 || die "Impossible d'ouvrir fichier: $!";

open(STDOUT, "> sortie")
 || die "Impossible d'ouvrir sortie: $!";
```

Et ces nouvelles E/S standard peuvent être accédées directement ou passées à d'autres processus. Tout se passe comme si le programme était appelé au départ avec ces redirections indiquées en ligne de commande.

Il est probablement plus intéressant de connecter les E/S standard à des tubes. Par exemple :

```
$pager = $ENV{PAGER} || "(less || more)";
open(STDOUT, "| $pager")
 || die "Impossible de forker un pager: $!";
```

Tout se passe comme si votre programme était appelé avec son stdout déjà redirigé vers votre pager. Vous pouvez aussi utiliser ce genre de choses en conjonction avec un fork implicite vers le script en cours. Vous pouvez avoir envie de faire ce genre de choses pour que votre programme gère lui-même le post traitement des données, mais dans un second processus :

```
head(100);
while (<>) {
 print;
}

sub head {
 my $lines = shift || 20;
 return if $pid = open(STDOUT, "|-"); # return si process père
 die "Échec du fork: $!" unless defined $pid;
 while (<STDIN>) {
 last if --$lines < 0;
 print;
 }
 exit;
}
```

Cette technique peut être itérée pour mettre en place autant de filtres successifs que vous le désirez sur le flux de sortie.

## 32.5 Autres considérations sur les E/S

Les sujets qui suivent ne sont pas directement liées aux paramètres de `open` ou de `sysopen`, mais ils agissent sur le comportement de vos fichiers ouverts.

### 32.5.1 Ouvrir des fichiers qui n'en sont pas

Quand un fichier n'est-il pas un fichier ? Une réponse possible consiste à dire qu'il existe mais que ce n'est pas un simple fichier. Nous vérifions tout d'abord s'il ne s'agit pas d'un lien symbolique, juste au cas où.

```
if (-l $file || ! -f _) {
 print "$file n'est pas un vrai fichier\n";
}
```

Quels sont les autres types de fichiers que (sic) les fichiers eux-mêmes ? Les répertoires, les liens symboliques, les tubes nommés, les sockets du domaine Unix et les périphériques blocs ou caractères. Tous sont des fichiers, mais pas de *simples* fichiers. Attention à ne pas tout confondre : la question n'est pas de savoir si ce sont des fichiers textes ou binaires. Il existe des fichiers textes qui ne sont pas de simples fichiers, et, inversement, il existe des fichiers simples qui ne sont pas des fichiers textes. C'est la raison pour laquelle il existe deux opérateurs unaires distincts de tests de fichiers `-f` et `-T`.

Pour ouvrir un répertoire, vous devriez utiliser la fonction `opendir`, puis la traiter avec `readdir`, en reconstruisant prudemment le chemin d'accès aux fichiers si nécessaire.

```
opendir(DIR, $dirname) or die "Échec d'opendir sur $dirname: $!";
while (defined($file = readdir(DIR))) {
 # faire quelque chose avec "$dirname/$file"
}
closedir(DIR);
```

Si vous voulez parcourir des dossiers récursivement mieux vaut utiliser le module `File::Find`. L'exemple suivant affiche tous les fichiers récursivement et ajoute un slash à leur nom si le fichier est un dossier.

```
@ARGV = qw(.) unless @ARGV;
use File::Find;
find sub { print $File::Find::name, -d && '/', "\n" }, @ARGV;
```

L'exemple suivant détecte tous les liens symboliques erronés dans l'arborescence d'un dossier particulier.

```
find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

Comme vous pouvez le constater avec les liens symboliques vous pouvez prétendre que le lien est ce vers quoi il pointe. Ou bien, si vous voulez savoir *vers quoi* il pointe, appeler `readlink`:

```
if (-l $file) {
 if (defined($pouf = readlink($fichier))) {
 print "$fichier pointe vers $pouf\n";
 } else {
 print "$fichier pointe dans le vide: $!\n";
 }
}
```

### 32.5.2 Ouvrir des tubes nommés

Les tubes nommés sont une toute autre affaire. On y accède comme s'il s'agissait de fichiers normaux, mais leur ouverture est généralement bloquante tant qu'il n'y a pas de candidats pour y écrire et pour y lire. Vous pouvez en apprendre plus sur leur fonctionnement en lisant *Tubes nommés* in *perlipc*. Les sockets du domaine Unix sont encore un autre genre de bestiole ; ils sont décrits dans *Clients et Serveurs TCP du Domaine Unix* in *perlfaq8*.

Quant à l'ouverture de fichiers de périphériques, elle est tantôt facile tantôt plus épineuse suivant le cas. Nous supposons que si vous ouvrez un périphérique de type "blocs", vous savez ce que vous êtes en train de faire. Les périphériques "caractères" méritent un peu plus d'attention. C'est ce type de périphérique qui est typiquement utilisé pour les modems et certains types d'imprimantes. C'est le sujet abordé dans *Comment lire et écrire sur des ports série ?* in *perlfaq8*. Il suffit en général de faire attention en les ouvrant :

```
sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
 # (O_NOCTTY n'est plus nécessaire sur les systèmes POSIX)
 or die "Impossible d'ouvrir /dev/ttyS1: $!";
open(TTYOUT, "+>&TTYIN")
 or die "Impossible de dupliquer TTYIN: $!";

$ofh = select(TTYOUT); $| = 1; select($ofh);

print TTYOUT "+++at\015";
$answer = <TTYIN>;
```

Pour les descripteurs que vous n'avez pas ouvert via `sysopen`, comme les sockets, il reste possible de forcer le mode non-bloquant en utilisant `fcntl`:

```
use Fcntl;
my $anciens_indicateurs = fcntl($handle, F_GETFL, 0)
 or die "Impossible de récupérer les indicateurs: $!";
fcntl($handle, F_SETFL, $anciens_indicateurs | O_NONBLOCK)
 or die "Impossible de passer en mode non bloquant: $!";
```

Plutôt que de perdre votre temps en vous enlisant dans les spécificités d'une foule d'`ioctl`s tous différents, si vous envisagez de manipuler des terminaux, alias *tty*s, mieux vaut faire appel au programme `stty(1)` s'il existe sur votre système, ou à défaut utiliser l'interface portable POSIX. Pour comprendre de quoi il retourne, vous devrez lire la page de man de `termios(3)` qui décrit l'interface POSIX vers les périphériques `tty`, et enfin *POSIX*, qui décrit l'interface Perl permettant d'accéder aux fonctions POSIX. Il existe aussi quelques modules de plus haut niveau sur CPAN qui peuvent vous aider. Voyez `Term::ReadKey` et `Term::ReadLine`.

### 32.5.3 Ouvrir des Sockets

Que reste-t'il encore à ouvrir ? Pour ouvrir une connexion par sockets, vous n'utiliserez aucun des deux `open` de Perl. Voyez pour cela `Sockets : Communication Client/Server` in *perlpc*. Voici un exemple. Une fois que vous l'avez vous pouvez utiliser `FH` comme n'importe quel handle de fichier bidirectionnel.

```
use IO::Socket;
local *FH = IO::Socket::INET->new("www.perl.com:80");
```

Pour ouvrir une URL suivez scrupuleusement l'ordonnance du docteur, une bonne cuillerée du module de CPAN `LWP` matin et soir. Ici aucun handle de fichier mais il reste très facile de récupérer le contenu d'un document :

```
use LWP::Simple;
$doc = get('http://www.linpro.no/lwp/');
```

### 32.5.4 Fichiers binaires

La lourde hérédité de certains systèmes leur fait utiliser des modèles d'entrées/sorties que les âmes charitables qualifient de mal fichus et tortueux (d'autres disent simplement défectueux). Un fichier n'est pas un fichier, du moins pas du point de vue des librairies d'E/S standard du C. Sur ces vieux systèmes sur lesquels les bibliothèques C (mais pas le noyau) font la différence entre les flux textes et binaires, quelques acrobaties sont nécessaires pour ne pas avoir d'ennuis avec les fichiers. Sur ces malencontreux systèmes les sockets et les tubes sont ouverts d'emblée en mode binaire, et il n'existe à ce jour aucun moyen de faire autrement. Dans le cas des fichiers vous avez un peu plus de marge de manoeuvre :

Vous pouvez appliquer la fonction `binmode` aux handles concernés avant tout accès aux fichiers :

```
binmode(STDIN);
binmode(STDOUT);
while (<STDIN>) { print }
```

Une alternative, sur les systèmes qui le gèrent, consiste à transmettre à `sysopen` une option non standard pour ouvrir le fichier en mode binaire. La solution est équivalente à une ouverture normale du fichier suivie d'un appel à `binmode` sur le handle.

```
sysopen(BINDAT, "fichier.data", O_RDWR | O_BINARY)
|| die "Impossible d'ouvrir fichier.data: $!";
```

Une fois cela fait vous pouvez utiliser `read` et `print` sur ce handle sans crainte que la bibliothèque d'entrées/sorties non standard détruise vos données. Ce n'est pas folichon, mais les tares héréditaires le sont rarement. CP/M restera avec nous jusqu'à la fin des temps... et après.

Sur les systèmes utilisant des couches d'E/S exotiques, il s'avère que, étonnamment, même les E/S non bufferisées qui utilisent `sysread` et `syswrite` peuvent discrètement mutiler vos données dès que vous avez le dos tourné.

```
while (sysread(WHENCE, $buf, 1024)) {
 syswrite(WHITHER, $buf, length($buf));
}
```

Suivant les vicissitudes de votre plate-forme d'exécution, même ces appels peuvent exiger un détour préalable par `binmode` ou `O_BINARY`. Il est de notoriété publique que Unix, MacOS, Plan 9 et et Inferno ne posent aucun problème.

### 32.5.5 Verrouillage de fichiers

Dans un environnement multitâches, il est parfois nécessaire d'être prudent afin d'éviter les conflits avec d'autres processus qui voudraient réaliser des E/S sur les fichiers auxquels vous êtes justement en train d'accéder.

Sur des fichiers en lecture et en écriture vous aurez souvent besoin de verrous, respectivement partagés ou exclusifs. Vous pouvez aussi faire comme s'il n'existait que des verrous exclusifs.

N'utilisez jamais l'opérateur d'existence d'un fichier `-e $fichier` en tant qu'indicateur de verrouillage, il existe en effet dans ce cas un problème d'accès concurrent (*race condition*) entre le test d'existence du fichier et sa création. Il est

possible pour un autre processus de créer un fichier pendant la tranche de temps qui sépare votre test d'existence et votre tentative de création du fichier. Ici l'atomicité de l'opération est critique.

Le mécanisme de verrouillage la plus portable en Perl consiste à utiliser la simplissime fonction `flock`, qui est émulée sur les systèmes qui ne la supportent pas directement comme SysV et Windows. La sémantique sous-jacente peut affecter le fonctionnement d'ensemble, vous devriez donc chercher à savoir comment `flock` est implémentée sur le portage de Perl de votre système d'exploitation.

Le verrouillage d'un fichier *n'empêche pas* les autres processus d'accéder au fichier. Un verrou de fichier bloque uniquement les processus demandant à verrouiller ce fichier, et n'empêche en rien de réaliser des opérations d'E/S sur celui-ci. Donc dans la mesure où le système des verrous fonctionne de manière purement coopérative, si un processus les utilise mais que d'autres les ignorent, rien ne va plus.

Par défaut un appel à `flock` va bloquer jusqu'à ce qu'un verrou soit accordé. Une demande de verrou partagé est accordée dès lors qu'aucune demande de verrou exclusif n'est active. Un verrou exclusif n'est accordé que si aucun verrouillage d'aucune sorte n'est actif. Les verrous sont associés à des descripteurs de fichier et non à des noms de fichiers, vous ne pouvez donc pas verrouiller un fichier avant de l'avoir ouvert ni maintenir un verrou actif une fois le fichier fermé.

Voici comment obtenir un verrou bloquant partagé sur un fichier, typiquement pour des accès en lecture :

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< fichier") or die "Erreur d'ouverture de fichier: $!";
flock(FH, LOCK_SH) or die "Erreur de verrouillage de fichier: $!";
now read from FH
```

Vous pouvez aussi obtenir un verrou non bloquant en utilisant `LOCK_NB`.

```
flock(FH, LOCK_SH | LOCK_NB)
 or die "Erreur de verrouillage de fichier: $!";
```

L'exemple suivant montre comment obtenir un comportement plus convivial en avertissant l'utilisateur que vous allez bloquer :

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< fichier") or die "Erreur d'ouverture de fichier: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
 $| = 1;
 print "En attente du verrou...";
 flock(FH, LOCK_SH) or die "Erreur de verrouillage du fichier: $!";
 print "Verrouillé.\n"
}
maintenant lire sur FH
```

Pour obtenir un verrou exclusif, typiquement nécessaire pour l'écriture, un minimum de prudence s'impose. Nous ouvrons le fichier par `sysopen` pour qu'il soit verrouillé avant d'être vidé. Un équivalent non bloquant est possible en utilisant `LOCK_EX | LOCK_NB`.

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "fichier", O_WRONLY | O_CREAT)
 or die "Erreur d'ouverture de fichier: $!";
flock(FH, LOCK_EX)
 or die "Erreur de verrouillage de fichier: $!";
truncate(FH, 0)
 or die "Erreur de troncature de fichier: $!";
maintenant écrire dans le fichier
```

Finalement, pour satisfaire ces trop nombreux webmestres que nul n'est parvenu à dissuader de gâcher quantité de cycles machines pour faire fonctionner un petit gadget vaniteux baptisé "Compteur de visites", voici comment incrémenter un nombre à l'intérieur d'un fichier en toute sécurité :

```

use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "fichiercompteur", O_RDWR | O_CREAT)
 or die "Erreur d'ouverture de fichiercompteur: $!";
autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
 or die "Erreur de verrouillage en écriture de fichiercompteur: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
 or die "Erreur de positionnement au début de fichiercompteur: $!";
print FH $num+1, "\n"
 or die "Erreur d'écriture dans fichier compteur: $!";

truncate(FH, tell(FH))
 or die "Erreur de troncature de fichiercompteur: $!";
close(FH)
 or die "Erreur de fermeture de fichiercompteur: $!";

```

### 32.5.6 Couches d'entrées/sorties

Avec Perl 5.8.0 un nouveau paradigme d'entrées/sorties baptisé "PerlIO" a été introduit. Il s'agit d'une "tuyauterie" flam-bant neuve pour tous les événements d'E/S en Perl ; pour l'essentiel tout fonctionne exactement comme par le passé, mais PerlIO offre aussi quelques nouvelles possibilités, comme permettre de penser aux E/S en termes de filtres successifs.

Un filtre d'E/S ne se contente plus uniquement de déplacer les données mais peut également leur appliquer des transformations. Il peut s'agir de transformations de type compression/décompression, cryptage/décryptage, ou encore de transcodages entre différents jeux de caractères.

Une présentation complète de PerlIO sortirait du cadre de ce tutoriel, mais voici déjà quelques façons d'indiquer l'utilisation de filtres :

- En utilisant la forme à trois arguments de `open` on ajoute au second argument quelque chose de plus que les habituels '`<`', '`>`', '`>>`', '`|`' et leurs variantes, par exemple :

```
open(my $fh, "<:crlf", $nomfichier);
```
- En utilisant la forme à deux arguments de `binmode` cela donne :

```
binmode($fh, ":encoding(utf16)");
```

Pour une discussion en profondeur de PerlIO voir *PerlIO*; Pour une discussion en profondeur d'Unicode et des E/S voir *perluniintro*.

## 32.6 VOIR AUSSI

Les fonctions `open` et `sysopen` dans *perlfunc*, les pages de manuel `open(2)`, `dup(2)`, `fopen(3)` et `fdopen(3)` ainsi que la documentation POSIX.

## 32.7 AUTEUR ET COPYRIGHT

Copyright 1998 Tom Christiansen.

Cette documentation est libre ; elle peut être redistribuée et modifiée sous les mêmes termes que Perl.

Indépendamment de leur distribution, tous les exemples de code figurant dans ce fichier sont placés dans le domaine public. Vous êtes autorisés et encouragés à utiliser ce code dans vos programmes que ce soit pour votre amusement ou contre rémunération. Un commentaire dans le code précisant son origine est une marque de courtoisie, mais absolument pas obligatoire.

## 32.8 TRADUCTION

### 32.8.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **32.8.2 Traducteur**

Christophe Grosjean <christophe.grosjean@gmail.com>

### **32.8.3 Relecture**

Pas encore...

## **32.9 HISTORIQUE**

Première version : Sat Jan 9 08:09:11 MST 1999

Traduction initiale : Mer Nov 8 11:41:00 GMT 2006

# Chapitre 33

## perlpod

Le format Pod (plain old documentation), la bonne vieille documentation

### 33.1 DESCRIPTION

Pod est un langage de balise, simple à utiliser pour écrire de la documentation pour Perl lui-même ainsi que pour les programmes Perl et les modules Perl.

Des traducteurs existent pour convertir le Pod vers différents formats comme le texte brut, le HTML, les pages man et d'autres encore.

Le langage Pod reconnaît à la base trois sortes de paragraphes : les paragraphes ordinaires (§33.1.1), les paragraphes de commande (§33.1.3) et les paragraphes verbatim (§33.1.2).

#### 33.1.1 Les paragraphes ordinaires

La plupart des paragraphes d'une documentation sont des blocs de texte ordinaires, comme celui-ci. Il vous suffit de taper votre texte sans aucune marque particulière et avec une ligne vide avant et après. Lorsqu'il sera formaté, ce bloc subira une mise en forme minimale comme un redécoupage des lignes, probablement écrites dans une police à espacement proportionnelle, qui seront sans doute justifiées.

Dans les paragraphes ordinaires, vous pouvez utiliser des codes de mise en forme pour le **gras**, l'*italique*, le style `code`, les liens et autres. Ces codes sont expliqués dans la section Codes de mise en forme (§33.1.4), ci-dessous.

#### 33.1.2 Les paragraphes verbatim

Les paragraphes verbatim (mot pout mot) sont habituellement utilisés pour présenter des blocs de code ou d'autres bouts de texte qui ne requièrent aucune analyse, aucune mise en forme particulière et dont les lignes ne doivent pas être redécoupées.

Un paragraphe verbatim se distingue par son premier caractère qui doit être un espace ou une tabulation. (Et habituellement, chacune de ses lignes commence par des espaces ou des tabulations.) Il devrait être reproduit à l'identique, en supposant que les tabulations sont alignées sur 8 caractères. Il n'existe aucun code de mise en forme et, par conséquent, aucune possibilité de faire de l'italique ou quoi que ce soit d'autre. Un `\` est un `\` et rien d'autre.

#### 33.1.3 Les paragraphes de commande

Un paragraphe de commande est utilisé pour spécifier des traitements spéciaux sur des parties du texte comme les titres ou les listes.

Tous les paragraphes de commande (qui ne font habituellement qu'une seule ligne) commencent par le caractère « = », suivi d'un identificateur, suivi d'un texte arbitraire que la commande peut utiliser de la façon qui lui plaît. Les commandes actuellement reconnues sont



```
=pod
=head1 titre
=head2 titre
=head3 titre
=head4 titre
=over niveauindentation
=item texte
=back
=begin format
=end format
=for format
=encoding type
=cut
```

Voici en détail des explications pour chacune d'elles :

**=head1 Texte du titre**

**=head2 Texte du titre**

**=head3 Texte du titre**

**=head4 Texte du titre**

Les commandes head1 à head4 produisent des titres et head1 est le titre de plus haut niveau. Le texte qui suit la commande et qui constitue le reste du paragraphe est le contenu du titre. Par exemple :

```
=head2 Attributs des objets
```

Le texte "Attributs des objets" est ici le titre. (Notez que les niveaux head3 et head4 sont des ajouts récents qui ne seront pas reconnus par de vieux traducteurs de Pod.) Le texte du titre peut utiliser des codes de mise en forme comme dans :

```
=head2 Valeurs possibles pour C<$/>
```

Ces codes sont expliqués dans la section Codes de mise en forme (§33.1.4), ci-dessous.

**=over niveauindentation**

**=item texte...**

**=back**

Les commandes item, over, et back ont besoin d'un peu plus d'explications : « =over » débute une section destinée à créer une liste utilisant des commandes « =item », ou pour indenter un ou plusieurs paragraphes normaux. Utilisez « =back » à la fin de votre liste ou de votre groupe de paragraphes. L'option *niveauindentation* de « =over » indique le niveau d'indentation, généralement mesuré en em (où un em est la largeur d'un M de la police de base du document) ou en une unité comparable. Si l'option *niveauindentation* est omise, sa valeur par défaut est quatre. (Et certains traducteurs ignoreront cette valeur quelle qu'elle soit.) Dans le *texte* de =item texte... vous pouvez utiliser des codes de mise en forme comme par exemple :

```
=item Utilisation de C<$/> pour contrôler l'usage des tampons
```

Ces codes sont expliqués dans la section Codes de mise en forme (§33.1.4), ci-dessous.

Notez aussi les quelques règles basiques suivantes pour bien utiliser les sections « =over » ... « =back » :

- N'utilisez pas « =item » en dehors d'une section « =over » ... « =back ».
- La première chose qui suit une commande « =over » devrait être une commande « =item », sauf s'il n'y a vraiment aucun item dans cette section « =over » ... « =back ».
- N'utilisez pas de commande « =head*n* » dans une section « =over » ... « =back ».
- Et, sans doute le plus important, utilisez des items cohérents entre eux : soit ce sont tous des « =item \* » pour produire une liste à puces ; soit ils sont tous de la forme « =item 1 », « =item 2 », etc. pour produire une liste numérotée ; soit ils sont tous de la forme « =item truc », « =item bidule », etc. pour produire une liste de définitions.

Si vous commencez par une puce ou par un numéro, continuez de même, puisque les traducteurs se basent sur le premier « =item » pour choisir le type de liste.

**=cut**

Pour terminer un bloc Pod, utilisez une ligne vide puis une ligne commençant par =cut puis encore une ligne vide. Ceci informe Perl (et les traducteurs Pod) que c'est à cet endroit que le code Perl recommence. (La ligne vide avant le =cut n'est pas techniquement indispensable mais beaucoup de vieux traducteurs Pod en ont besoin.)

**=pod**

La commande `=pod` en elle-même ne sert pas à grand chose si ce n'est de signaler à Perl (et aux traducteurs Pod) qu'une section Pod commence à cet endroit. Une section Pod peut commencer par *n'importe* quel paragraphe de commande. Une commande `=pod` ne sert donc qu'à indiquer une section Pod qui débute directement par un paragraphe ordinaire ou un paragraphe verbatim. Par exemple :

```
=item trucs()

Cette fonction fait des trucs.

=cut

sub trucs {
 ...
}

=pod

Souvenez-vous de vérifier son S<résultat :>

 trucs() || die "Ne peux pas faire des trucs !";

=cut
```

**=begin *nomformat*****=end *nomformat*****=for *nomformat* *texte...***

Les commandes `for`, `begin` et `end` vous permettent d'utiliser des sections de texte/code/donnée qui ne seront pas interprétées comme du Pod normal mais qui pourront être utilisées directement par des traducteurs spécifiques ou qui pourront avoir un usage spécial. Seuls les traducteurs qui savent comment utiliser le format spécifié utiliseront cette section. Sinon elle sera complètement ignorée.

Une commande «`=begin nomformat`» puis quelques paragraphes et enfin une commande «`=end nomformat`» signifie que les paragraphes inclus sont réservés aux traducteurs comprenant le format spécial appelé *nomformat*. Par exemple :

```
=begin html

<hr>
<p>Ceci est un paragraphe HTML</p>

=end html
```

La commande «`=for nomformat texte...`» indique que c'est uniquement ce paragraphe (le *texte* qui est juste après *nomformat*) qui est dans ce format spécial.

```
=for html <hr>
<p>Ceci est un paragraphe HTML</p>
```

Les deux exemples ci-dessus produiront le même résultat.

La différence est qu'avec «`=for`», seul le paragraphe est concerné alors qu'entre le couple «`=begin format`» ... «`=end format`», vous pouvez placer autant de contenu que nécessaire. (Notez que les lignes vides après la commande `=begin` et avant la commande `=end` sont requises.)

Voici des exemples de l'utilisation de ceci :

```
=begin html

Figure 1.

=end html

=begin text

| foo |
| bar |
| |

^^^^ Figure 1. ^^^^
```

```
=end text
```

Parmi les noms de format actuellement connus pour être reconnus par les formateurs, on trouve « roff », « man », « latex », « tex », « text » et « html ». (Des traducteurs Pod peuvent en considérer certains comme synonymes.)

Le nom de format « comment » est pratique pour placer des notes (juste pour vous) qui n'apparaîtront dans aucune version mise en forme de la documentation Pod :

```
=for comment
```

S'assurer que toutes les options sont documentées !

Quelques *nomformats* utilisent le préfixe `:` (comme par exemple `=for :nomformat` ou `=begin :nomformat ... =end :nomformat`) pour indiquer que le texte ne doit pas être considéré comme brut mais qu'il contient en fait du texte Pod (c.-à-d. contenant éventuellement des codes de mise en forme) qui n'est pas à utiliser pour une mise forme normale (c.-à-d. qui n'est pas un paragraphe normal mais pourrait être utilisé, par exemple, comme note de bas de page).

### **=encoding *codage***

Cette commande permet de déclarer le codage utilisé dans le document. La plupart des utilisateurs n'en ont pas besoin ; mais si votre encodage n'est pas US-ASCII ou Latin-1 alors indiquez-le aux traducteurs Pod en plaçant une commande `=encoding codage` le plus tôt possible dans le document. Comme *codage*, utilisez l'un des noms reconnus par le module `Encode::Supported`. Exemples :

```
=encoding utf8
```

```
=encoding koi8-r
```

```
=encoding ShiftJIS
```

```
=encoding big5
```

N'oubliez pas, en utilisant une commande, que cette commande se termine à la fin de son *paragraphe*, pas à la fin de sa ligne. D'où dans les exemples ci-dessus, les lignes vides que vous pouvez voir après chaque commande pour terminer son paragraphe.

Quelques exemples de listes :

```
=over
```

```
=item *
```

```
Premier item
```

```
=item *
```

```
Second item
```

```
=back
```

```
=over
```

```
=item Foo()
```

```
Description de la fonction Foo
```

```
=item Bar()
```

```
Description de la fonction Bar
```

```
=back
```

### 33.1.4 Codes de mise en forme

Dans les paragraphes ordinaires et dans certains paragraphes de commande, plusieurs codes de mise en forme (appelés aussi « séquences internes») peuvent être utilisés :

#### I<texte> – texte en italique

Utilisé pour mettre en évidence (faites I<attention !>) et pour les paramètres (redo I<LABEL>).

#### B<texte> – texte en gras

Utilisé pour les options (l'option B<-n> de perl), pour les programmes (certains systèmes proposent B<chfn> pour ça), pour mettre en évidence (faites B<attention !>) et autres (cette propriété s'appelle B<l'autovivification>).

#### C<code> – du code

Présente le code dans une police type machine à écrire ou donne une indication que le texte est un programme (C<gmtime(\$^T)>) ou quelque chose liée à l'ordinateur (C<drwxr-xr-x>).

#### L<nom> – un hyperlien

Il y a différentes syntaxes, présentées ci-dessous. Dans ces syntaxes, `texte`, `nom` et `section` ne peuvent pas contenir les caractères `'/` et `'|` et les caractères `'<` ou `'>` doivent pouvoir s'associer.

– L<nom>

Lien vers une page de documentation Perl (par exemple L<Net::Ping>). Notez que `nom` ne devrait pas contenir d'espaces. Cette syntaxe est aussi utilisé occasionnellement pour faire référence aux pages de manuel UNIX comme dans L<crontab(5)>.

– L<nom/"section"> ou L<nom/section>

Lien vers une section particulière d'une autre page de documentation. Par exemple L<perlsyn/"Boucles for">.

– L</"section"> ou L</section> ou L<"section">

Lien vers une section particulière de ce même document. Par exemple L</"Méthodes objets">.

Une section débute par un titre ou un item. Par exemple, L<perlvar/\$.> ou L<perlvar/"\$. "> seront tous deux liés à la section qui débute par `=item $.` dans `perlvar`. Et L<perlsyn/Boucles for> ou L<perlsyn/"Boucles for"> sont tous deux liés à la section débutant par `=head2 Boucle for` dans `perlsyn`.

Pour contrôler le texte affiché comme lien, vous pouvez utiliser L<texte|...> comme dans :

– L<texte|nom>

Lié ce texte à la page de documentation dont le nom est fourni. Par exemple L<Messages d'erreurs de Perl|perldiag>.

– L<texte|nom/"section"> ou L<texte|nom/section>

Lié ce texte à la section de la page de documentation dont le nom est fourni. Par exemple L<postfix "if"|perlsyn/"Statement Modifiers">

– L<texte|/"section"> or L<texte|/section> ou L<texte|"section">

Lié ce texte à la section de ce même document. Par exemple L<les différents attributs|/"Données membres">

Ou vous pouvez lier une page web :

– L<scheme:...>

Lien vers un URL absolu. Par exemple L<http://www.perl.org/>. Mais notez que, pour différentes raisons, il n'existe pas de syntaxe du genre L<text|scheme:...>.

#### E<entité> – un caractère nommé

Très similaire aux « entités » HTML/XML &foo;

– E<lt> – un < littéral (plus petit que)

– E<gt> – un > littéral (plus grand que)

– E<verbar> – un | littéral (*barre verticale*)

– E<sol> – un / littéral (*barre oblique*)

Les quatre codes ci-dessus sont optionnels sauf s'ils sont utilisés à l'intérieur d'un autre de code de mise en forme, en particulier L<...> ou lorsqu'ils sont directement précédés d'une lettre majuscule.

– E<htmlentité>

Quelques entités HTML non numériques telle que E<eacute>, qui signifie la même chose que &eacute; ; en HTML – c.-à-d. un e minuscule avec un accent aigu.

– E<nombre>

Le caractère ASCII/Latin-1/Unicode dont le code est le nombre. Le préfixe "0x" indique que *nombre* est en hexadécimal comme dans E<0x201E>. Le préfixe "0" indique que le nombre est en octal comme dans E<075>. Sinon, le *nombre* est considéré en décimal comme dans E<181>.

Notez que les vieux traducteurs Pod peuvent ne pas reconnaître l'octal et l'hexadécimal et que de nombreux traducteurs ne savent pas présenter correctement les caractères dont le code est supérieur à 255. (Certains traducteurs peuvent même choisir un compromis pour présenter les caractères Latin-1, en présentant un simple "e" à la place de E<eacute>.)

**F<nomfichier> – utilisé pour un nom de fichier**

Typiquement affiché en italique. Exemple : F<.cshrc>

**S<texte> – texte contenant des espaces non sécables**

Cela signifie que les mots du *texte* ne doivent pas être séparés sur plusieurs lignes. Exemple : S<\$x ? \$y : \$z>.

**X<nom de sujet> – une point d'entrée d'index**

Ce code est ignoré par la plupart des traducteurs mais certains peuvent l'utiliser pour construire un index. Il est toujours présenté comme la chaîne vide. Exemple : X<URL absolu>

**Z<> – un code de mise en forme nul (sans effet)**

C'est rarement utilisé. C'est l'un des moyens pour empêcher l'interprétation d'un code E<...>. Par exemple, à la place de "NE<lt>3" (pour "N<3"), vous pourriez écrire "NZ<><3" (le code "Z<>" sépare le "N" et le "<" afin qu'ils ne soient pas considérés comme le début d'une (hypothétique) séquence "N<...>").

La plupart du temps, un simple couple inférieur/supérieur suffira pour délimiter le début et la fin de votre code de mise en forme. Mais il se peut que vous ayez besoin de placer un symbole supérieur (un signe « plus grand que », '>') dans un code de mise en forme. C'est très courant lorsqu'on souhaite présenter un extrait de code avec une police différente du reste du texte. Comme d'habitude en Perl, il y a plusieurs moyens pour le faire. Le premier consiste tout simplement à utiliser l'entité « plus grand que » via le code de mise en forme E :

```
C<$a E<lt>=E<gt> $b>
```

Ce qui produira : \$a <=> \$b.

Un moyen plus lisible et probablement plus "brut" est d'utiliser de délimiteurs qui n'imposent pas de codage spécial pour un simple ">". Les traducteurs Pod proposent cela en standard depuis perl 5.5.660 via les doubles délimiteurs ("<<" et ">>") qui peuvent être utilisés *si et seulement si il y a un espace après le délimiteur ouvrant et un espace avant le délimiteur fermant!*. Par exemple :

```
C<< $a <=> $b >>
```

En fait, vous pouvez utiliser le nombre d'inférieurs et de supérieurs que vous souhaitez tant qu'il y en a autant dans le délimiteur ouvrant que dans le délimiteur fermant et si vous vous assurez qu'un espace suit immédiatement le dernier « < » du délimiteur ouvrant et qu'un autre espace précède immédiatement le premier « > » du délimiteur fermant (ces espaces seront ignorés). Les exemples suivants fonctionneront donc aussi :

```
C<<< $a <=> $b >>>
C<<<< $a <=> $b >>>>
```

Et ils signifient exactement la même chose que :

```
C<$a E<lt>=E<gt> $b>
```

Prenons un autre exemple : supposons que vous voulez présenter l'extrait de code suivant dans un style C<> (code) :

```
open(X, ">>thing.dat") || die $!
$foo->bar();
```

Vous pourrez le faire comme cela :

```
C<<< open(X, ">>thing.dat") || die $! >>>
C<< $foo->bar(); >>
```

qui est certainement plus facilement lisible que l'ancien méthode :

```
C<open(X, "E<gt>E<gt>thing.dat") || die $!>
C<$foo-E<gt>bar();>
```

Tout cela est actuellement accepté par pod2text (Pod::Text), pod2man (Pod::Man) et tout autre traducteur pod2xxx et Pod::Xxxx qui utilise Pod::Parser version 1.093 ou supérieure.

### 33.1.5 L'objectif

L'objectif est la simplicité d'utilisation, pas la puissance expressive. Les paragraphes ont l'air de paragraphes (des blocs) pour qu'ils ressortent visuellement, et on peut les faire passer facilement à travers `fmt` pour les reformater (c'est F7 dans ma version de `vi` ou Esc-Q dans ma version de `emacs`). Je voulais que le traducteur laisse les `'`, les `'` et les `"` tranquilles en mode verbatim pour que je puisse copier/coller ces paragraphes dans un programme qui marche, les décaler de 4 espaces, et l'imprimer, euh, mot pour mot. Et probablement dans une fonte à chasse fixe.

Le format Pod est certainement insuffisant pour rédiger un livre. Pod essaie juste d'être un format infaillible qui puisse servir de source pour `nroff`, HTML, TeX et autres langages de balises, lorsqu'ils sont utilisés pour de la documentation en ligne. Des traducteurs existent pour **pod2text**, **pod2html**, **pod2man** (c'est pour `nroff(1)` et `troff(1)`), **pod2latex** et **pod2fm**. D'autres encore sont disponibles sur CPAN.

### 33.1.6 Incorporer du Pod dans les modules Perl

Vous pouvez inclure de la documentation Pod dans vos scripts et vos modules Perl. Commencez votre documentation par une ligne vide puis une commande `=head1` et terminez-la par une commande `=cut` et une ligne vide. Perl ignorera le texte en Pod. Regardez n'importe lequel des modules fournis en standard pour vous servir d'exemple. Si vous souhaitez mettre votre Pod à la fin du fichier, et si vous utilisez un `__END__` ou un `__DATA__` comme marque de fin, assurez-vous de mettre une ligne vide avant votre première commande Pod.

```
__END__
```

```
=head1 NAME
```

```
Time::Local - efficiently compute time from local and GMT time
```

Sans cette ligne vide avant `=head1`, de nombreux traducteurs ne reconnaîtront pas `=head1` comme le début d'une section Pod.

### 33.1.7 Conseils pour écrire en Pod

- La commande **podchecker** permet de vérifier le respect de la syntaxe Pod. Par exemple, elle vérifie les lignes entièrement blanches dans les sections Pod ou les commandes et les codes de mise en forme inconnus. Vous pouvez aussi passer votre document au travers d'un ou plusieurs traducteurs Pod et vérifier le résultat (en l'imprimant si besoin est). Certains problèmes rencontrés peuvent être liés à des bogues des traducteurs. À vous de décider si vous voulez les contourner ou non.
- Si vous êtes plus à votre aise en rédigeant du HTML que du Pod, vous pouvez rédiger votre documentation en HTML simple puis la convertir en Pod grâce au module expérimental `Pod::HTML2Pod` (disponible sur CPAN) et enfin vérifier le code obtenu. Le module expérimental `Pod::PXML` peut aussi être utile.
- De nombreux traducteurs Pod ont absolument besoin d'une ligne blanche avant et après chaque commande Pod (commande `=cut` y compris). Quelque chose comme ça :

```
- - - - -
=item $firecracker->boom()
This noisily detonates the firecracker object.
=cut
sub boom {
...

```

...amènera ces traducteurs Pod à ignorer totalement la section Pod.

À la place, préférez quelque chose comme ceci :

```
- - - - -
=item $firecracker->boom()
This noisily detonates the firecracker object.
=cut
sub boom {
...

```

- Certains traducteurs Pod anciens ont absolument besoin de paragraphes (incluant les paragraphes de commande comme `"=head2 Fonctions"`) séparés par des lignes *complètement* vides. Si vous avez des lignes apparemment vides mais contenant en fait des espaces, elles ne seront pas reconnues comme séparateurs par ces traducteurs et provoqueront peut-être de mauvaises mises en forme.

- Des traducteurs Pod anciens ajoutent quelques mots autour de certains liens L<> de telle manière que L<Foo::Bar> deviendra par exemple "the Foo::Bar manpage". Vous ne pouvez donc pas écrire des choses telles que la documentation L<bidule> si vous voulez que le résultat reste compréhensible. À la place, écrivez la documentation L<bidule|bidule> ou L<la documentation bidule|bidule> pour contrôler l'apparence du lien.
- Un texte qui dépasse la 70e colonne dans un bloc verbatim peut être arbitrairement coupé par certains traducteurs.

## 33.2 VOIR AUSSI

*perlpodspec*, POD: documentation intégrée in *perlsyn*, *perlnewmod*, *perldoc*, *pod2html*, *pod2man*, *podchecker*.

## 33.3 AUTEUR

Larry Wall, Sean M. Burke

## 33.4 TRADUCTION

### 33.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 33.4.2 Traducteur

Traduction initiale : Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 33.4.3 Relecture

Gérard Delafond

# Chapitre 34

## perlrun

Comment utiliser l'interpréteur Perl

### 34.1 SYNOPSIS

```
perl [-CsTuWX] [-hv] [-V[:configvar]] [-cw] [-d[:debugger]] [-D[number/list]] [-pna] [-Fpattern] [-I[octal]] [-O[octal/hexadecimal]] [-Idir] [-m[-]module] [-M[-]'module...'] [-f] [-C [number/list]] [-P] [-S] [-x[dir]] [-i[extension]] [-e 'command'] [-] [programfile] [argument]...
```

### 34.2 DESCRIPTION

La façon habituelle d'exécuter un programme Perl est de le rendre directement exécutable ou bien de passer le nom du fichier source comme argument de ligne de commande (Un environnement Perl interactif est aussi possible – voir *perldebug* pour plus de détails sur son utilisation). Au lancement, Perl recherche votre script dans l'un des endroits suivants :

1. Spécifié ligne par ligne par l'intermédiaire d'options **-e** sur la ligne de commande.
2. Contenu dans le fichier spécifié par le premier nom de fichier sur la ligne de commande. (Notez que les systèmes qui supportent la notation **#!** invoquent les interpréteurs de cette manière. Cf. Emplacement de Perl.)
3. Passé implicitement sur l'entrée standard. Cela ne marche que s'il n'y a pas de noms de fichiers comme argument – pour passer des arguments à un programme lisant sur STDIN, vous devez explicitement spécifier un **"-"** pour le nom du script.

Avec les méthodes 2 et 3, Perl démarre l'analyse du fichier d'entrée à partir du début, à moins de rajouter une option **-x**. Dans ce cas, il cherche la première ligne commençant par **#!** et contenant le mot "perl". L'interprétation commence alors à partir de cette ligne. C'est utile pour lancer un programme contenu dans un message plus grand. (Dans ce cas, la fin du programme doit être indiquée par **\_\_END\_\_**.)

Lorsque la ligne commençant par **#!** est analysée, les éventuelles options sont toujours recherchées. Ainsi, si vous êtes sur une machine qui n'autorise qu'un seul argument avec la ligne **#!**, ou pire, qui ne reconnaît même pas la ligne **#!**, vous pouvez toujours obtenir un comportement homogène vis-à-vis des options, quelle que soit la manière dont Perl a été invoqué, même si **-x** a été utilisé pour trouver le début du programme.

Puisque historiquement certains systèmes d'exploitation arrêtaient l'interprétation de la ligne **#!** par le noyau à partir de 32 caractères, certaines options peuvent être passées sur la ligne de commande, d'autres pas ; vous pouvez même vous retrouver avec un **"-"** sans sa lettre au lieu d'une option complète, si vous n'y faites pas attention. Vous devrez probablement vous assurer que toutes vos options tombent d'un côté ou de l'autre de la frontière des 32 caractères. La plupart des options ne se soucient pas d'être traitées de façon redondante, mais obtenir un **"-"** au lieu d'une option complète pourrait pousser Perl à essayer d'exécuter l'entrée standard au lieu de votre programme. Une option **-I** incomplète pourrait aussi donner des résultats étranges.

Certaines options sont sensibles au nombre de fois où elles sont présentes sur la ligne de commande, par exemple, les combinaisons des options **-I** et **-O**. Il faut soit mettre toutes les options après la limite des 32 caractères (si possible), soit remplacer les utilisations de **-Odigits** par **BEGIN{ \$/ = "\0digits"; }**.

L'analyse des options commence dès que "perl" est mentionné sur la ligne. Les séquences **"-\*"** et **"- "** sont spécialement ignorées de telle manière que vous puissiez écrire (si l'envie vous en prend), disons



```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl -wS $0 ${1+"$@"}'
 if $running_under_some_shell;
```

pour permettre à Perl de voir l'option **-p**.

Un truc similaire implique le programme **env**, si vous le possédez.

```
#!/usr/bin/env perl
```

Les exemples ci-dessus utilisent un chemin relatif vers l'interpréteur perl et utilisent donc la première version trouvée dans le PATH de l'utilisateur. Si vous voulez une version spécifique de Perl, disons, perl5.005\_57, vous devez le placer directement dans le chemin de la ligne #!.

Si la ligne #! ne contient pas le mot "perl", le programme indiqué après le #! est lancé au lieu de l'interpréteur Perl. C'est assez bizarre, mais cela aide les gens qui utilisent des machines qui ne reconnaissent pas #!, car ainsi ils peuvent dire à un programme que leur SHELL est */usr/bin/perl*, et Perl enverra le programme vers le bon interpréteur pour eux.

Après avoir localisé votre programme, Perl le compile en une forme interne. Si il y a ne serait-ce qu'un erreur de compilation, l'exécution du programme n'est pas tentée (c'est différent d'un script shell classique, qui peut être en partie exécuté avant que soit détectée une erreur de syntaxe).

Si le programme est syntaxiquement correct, il est exécuté. Si le programme se termine sans rencontrer un opérateur die() ou exit(), un exit(0) implicite est ajouté pour indiquer une exécution réussie.

### 34.2.1 #! et l'isolement (quoting) sur les systèmes non-Unix

La technique #! d'Unix peut être simulée sur les autres systèmes :

#### OS/2

Ajoutez

```
extproc perl -S -vos_options
```

sur la première ligne de vos fichiers \*.cmd (le -S est dû à un bug dans la manière dont cmd.exe gère 'extproc').

#### MS-DOS

Créez un fichier de commande (batch) pour lancer votre programme, et placez son nom dans ALTERNAT\_SHEBANG ("#!" se prononce She Bang, cf. Jargon File, NDT). Pour plus d'informations à ce sujet, voir le fichier *dosish.h* dans les sources de la distribution.

#### Win95/NT

L'installation Win95/NT, en tout cas l'installateur Activestate pour Perl, va modifier la "base de registre" pour associer l'extension *.pl* avec l'interpréteur Perl. Si vous installez Perl par d'autres moyens (y compris via une compilation à partir des sources), vous devrez modifier la base de registre vous-même. Notez que cela implique que vous ne pourrez plus faire la différence entre un exécutable Perl et une bibliothèque Perl.

#### Macintosh

Dans MacOS "Classic", les programmes Perl auront le Creator et le Type adéquat, donc un double-clic lancera l'application MacPerl. Dans Mac OS X, un script avec #! est transformable en une application double-clicable en utilisant l'utilitaire DropScript de Wil Sanchez : <http://www.wsanchez.net/software/>.

#### VMS

Ajoutez

```
$ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

au début de votre programme, où *-mysw* représente toutes les options de ligne de commande que vous voulez passer à Perl. Vous pouvez désormais invoquer le programme directement, en disant `perl programme`, ou en tant que procédure DCL, en disant `@programme` (ou implicitement via `DCL$PATH` en utilisant juste le nom du programme). Cette incantation est un peu difficile à mémoriser, mais Perl l'affichera pour vous si vous dites `perl "-V:startperl"`.

Les interpréteurs de commandes des systèmes non-Unix ont des idées plutôt différentes des shells Unix concernant l'isolement (quoting). Il vous faudra apprendre quels sont les caractères spéciaux de votre interpréteur de commandes (\*, \ et " le sont couramment), et comment protéger les espaces et ces caractères pour lancer des one-liners [Ndt] scripts sur une seule ligne [Fin de Ndt] (cf. `-e` plus bas).

Sur certains systèmes, vous devrez peut-être changer les apostrophes (single-quotes) en guillemets (double quotes), ce qu'il ne faut *pas* faire sur les systèmes Unix ou Plan9. Vous devrez peut-être aussi changer le signe % seul en %%.

Par exemple :

```
Unix
perl -e 'print "Hello world\n"'

MS-DOS, etc.
perl -e "print \"Hello world\n\""

Macintosh
print "Hello world\n"
 (puis Run "Myscript" ou Shift-Command-R)

VMS
perl -e "print ""Hello world\n"""
```

Le problème, c'est que le fonctionnement de ces exemples n'est absolument pas garanti : cela dépend de la commande et il est possible qu'aucune forme ne fonctionne. Si l'interpréteur de commande était **4DOS**, ceci marcherait probablement <mieux :>

```
perl -e "print <Ctrl-x>\"Hello world\n<Ctrl-x>\""
```

**CMD.EXE** dans Windows NT a récupéré beaucoup de fonctionnalités Unix lorsque tout le monde avait le dos tourné, mais essayez de trouver de la documentation sur ses règles d'isolement !

Sur Macintosh, cela dépend de l'environnement que vous utilisez. Le shell MacPerl, ou MPW, est très proche des shells Unix pour son support de différentes variantes d'isolement, si ce n'est qu'il utilise allègrement les caractères Macintosh non ASCII comme caractères de contrôle.

Il n'y a pas de solution générale à ces problèmes. C'est juste le foutoir.

### 34.2.2 Emplacement de Perl

Cela peut sembler évident, mais Perl est utile lorsque les utilisateurs y ont facilement accès. Autant que possible, il est bon d'avoir `/usr/bin/perl` et `/usr/local/bin/perl` comme liens symboliques (symlinks) vers le bon binaire. Si cela n'est pas possible, nous encourageons les administrateurs systèmes à mettre perl et les utilitaires associés (directement ou sous forme de liens symboliques) dans un répertoire présent dans le PATH des utilisateurs, ou encore dans un quelconque autre endroit pratique et évident.

Dans cette documentation, `#!/usr/bin/perl` dans la première ligne d'un programme, indiquera ce qui marche sur votre système. Nous vous conseillons d'utiliser un chemin spécifique si vous avez besoin d'une version particulière.

```
#!/usr/local/bin/perl5.00554
```

ou si vous voulez juste utiliser une version supérieure ou égale à une version particulière, placez une instruction telle que celle-ci au début de votre programme :

```
use 5.005_54;
```

### 34.2.3 Options de ligne de commandes

Comme pour toutes les commandes standard, une option mono-caractère peut être combinée avec l'option suivante, le cas échéant.

```
#!/usr/bin/perl -spi.orig # équivalent à -s -p -i.orig
```

Les options comprennent :

#### **-0**[*octal/hexadecimal*]

indique le séparateur d'enregistrements en entrée (\$/) en notation octale ou hexadécimale. S'il n'y a pas de chiffres, le caractère nul (ASCII 0) est le séparateur. D'autres options peuvent suivre ou précéder les chiffres. Par exemple, si vous avez une version de **find** qui peut afficher les noms de fichiers terminés par des caractères nuls, vous pouvez écrire ceci :

```
find . -name '*.orig' -print0 | perl -n0e unlink
```

La valeur spéciale 00 va indiquer à Perl d'avaler les fichiers en mode paragraphes. La valeur 0777 indique à Perl d'avaler les fichiers en entier car il n'y a pas de caractères avec cette valeur octale.

Si vous voulez spécifier un caractère Unicode, utilisez la notation hexadécimale : **-0xHHH...**, où les H représentent des chiffres hexadécimaux. (Cela signifie que vous ne pouvez pas utiliser l'option **-x** avec un nom de répertoire constituée uniquement de chiffres hexadécimaux.)

#### **-a**

active le mode auto-découpage (autosplit) lorsqu'utilisé avec **-n** ou **-p**. On insère une commande split implicite vers le tableau @F au début de la boucle while implicite produite par **-n** ou **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

est équivalent à :

```
while (<>) {
 @F = split(' ');
 print pop(@F), "\n";
}
```

Un autre séparateur [Ndt] que espace ' '[Fin Ndt] peut être spécifié via **-F**.

#### **-C**[*nombre/liste*]

L'option **-C** contrôle les fonctionnalités Unicode de Perl.

Depuis la version 5.8.1, le **-C** peut être suivi par un nombre ou une suite de lettres. Les lettres, leur valeur numérique et leur effet sont les suivants (une liste de lettre est équivalente à la somme de leur valeur) :

I	1	STDIN est supposé être en UTF-8
O	2	STDOUT sera en UTF-8
E	4	STDERR sera en UTF-8
S	7	I + O + E
i	8	UTF-8 est le filtre PerlIO par défaut pour les flux entrants
o	16	UTF-8 est le filtre PerlIO par défaut pour les flux sortants
D	24	i + o
A	32	les éléments de @ARGV sont des chaînes codées en UTF-8
L	64	normalement "IOEioA" s'appliquent sans condition.

Le L conditionne leur prise en compte aux variables d'environnement (LC\_ALL, LC\_TYPE et LANG, par ordre de priorité décroissant). Si ces variables indiquent UTF-8 alors les options "IOEioA" sont prises en compte.

Par exemple, **-COE** ou **-C6** activent l'UTF-8 sur STDOUT et STDERR. Des lettres répétées sont juste redondantes. Elles ne se cumulent pas ou ne s'annulent pas.

Les options **io** indiquent que tous les **open()** (ou autres opérations d'E/S similaires) utiliseront implicitement le filtre PerlIO **:utf8**. En d'autres termes, tous les flux entrants seront lus en UTF-8 et tous les flux sortants seront écrits en UTF-8. Ce n'est que le comportement par défaut puisqu'il est toujours possible de changer ce comportement en indiquant explicitement les filtres voulus dans les appels à **open()** ou à **binmode()**.

L'option **-C** seule ou la chaîne vide "" pour la variable d'environnement PERL\_UNICODE ont le même effet que **-CSDL**. En d'autres termes, les flux d'E/S standard et le filtre par défaut de **open()** sont UTF-8 **uniquement** si les

variables d'environnement demandent un locale UTF-8. Ce comportement reproduit le comportement implicite (en problématique) de Perl 5.8.0 vis-à-vis d'UTF-8.

Vous pouvez utiliser `-C0` (ou `"0"` pour `PERL_UNICODE`) pour désactiver explicitement toutes les fonctionnalités Unicode ci-dessus.

La variable magique `${^UNICODE}`, accessible uniquement en lecture, reflète la valeur numérique de ces réglages. Cette variable est calculée lors du démarrage de Perl et n'est plus modifiable par la suite. Si vous voulez changer de comportement dynamiquement, utilisez `open()` avec trois arguments (voir `open` in *perlfunc*), `binmode()` avec deux arguments (voir `binmode` in *perlfunc*) ou la directive `open` (voir *open*).

(Dans les versions de Perl antérieures à la version 5.8.1, l'option `-C` n'existait que sur Win32 et permettait d'activer les "wide system call" Unicode de l'API Win32. Cette fonctionnalité n'était quasiment pas utilisée et l'option a donc été "recyclée".)

**-c**

demande à Perl de vérifier la syntaxe du programme puis de quitter sans l'exécuter. En fait, il va exécuter les blocs BEGIN et CHECK ainsi que les instructions use car ils sont considérés comme se déroulant avant l'exécution de votre programme. En revanche, les blocs INIT et END ne seront pas exécutés.

**-d****-dt**

lance le programme sous le débogueur (debugger) Perl. Cf. *perldebug*. Si le `t` est présent, cela indique au débogueur que les fils d'exécution (les threads) sont utilisés dans le code à déboguer.

**-d:foo[=bar,baz]****-dt:foo[=bar,baz]**

lance le programme sous le contrôle du module de traçabilité, de profilage ou de débogage installé sous le nom `Devel::foo`. C.-à-d. que `-d:DProf` exécute le programme en utilisant le profileur `Devel::DProf`. Comme pour l'option `-M`, on peut passer des options au package `Devel::foo` et elles seront reçues et interprétées par la routine `Devel::foo::import`. La liste d'options séparées par des virgules doit être placée après le caractère `=`. Si le `t` est présent, cela indique au débogueur que les fils d'exécution (les threads) sont utilisés dans le code à déboguer. Voir *perldebug*.

**-Dlettres****-Dnombre**

détermine les drapeaux (flags) de débogage de Perl. Pour voir comment perl exécute votre programme, utilisez `-Dtls`. (Cela ne fonctionne que si le support du débogage est compilé dans votre interpréteur Perl). Une autre valeur intéressante est `-Dx`, qui affiche l'arbre syntaxique compilé. Et `-Dr` affiche les expressions rationnelles compilées ; le format de cette sortie est expliqué dans *perldebug*.

Vous pouvez fournir un nombre à la place d'une listes de lettres (par exemple, `-D14` est équivalent à `-Dtls`):

```

1 p Découpage en unités lexicales et analyse
2 s Clichés de la pile (avec v, affiche toutes les piles)
4 l Traitement des piles de contextes de boucles
 (Context (loop) stack processing)
8 t Exécution tracée (Trace execution)
16 o Résolution et surcharge de méthodes
32 c Conversions chaînes/nombres
64 P Affiche les commandes du pré-processeur pour -P
128 m Allocation mémoire
256 f Traitement des formats
512 r Analyse et exécution des expressions rationnelles
1024 x Affichage de l'arbre syntaxique
2048 u Vérification des données souillées (tainted)
4096 (Obsolète, utilisé auparavant pour les fuites mémoire)
8192 H Affiche les tables de hachage (usurps values())
16384 X Allocation scratchpad
32768 D Nettoyage
65536 S Synchronisation des fils d'exécution
131072 T Découpage en unités lexicales
262144 R Inclure le compteur de références des variables affichés
 (utilisé avec -Ds)
524288 J Ne pas s,t,P-déboguer (Sauter) les opcodes du package DB
1048576 v Verbose: à utiliser avec d'autres drapeaux
8388608 q muet - pour l'instant, ne supprime que le message "EXECUTING"

```

Tous ces drapeaux nécessitent **-DDEBUGGING** quand vous compilez l'exécutable Perl (mais voyez tout de même *Devel::Peek* et *re* qui peuvent changer cela). Voir le fichier *INSTALL* dans la distribution des sources de Perl pour savoir comment le faire. Ce drapeau est automatiquement rajouté si vous compilez avec l'option **-g** quand *Configure* vous demande les drapeaux de votre optimiseur/débugueur.

Si vous essayez juste d'obtenir l'affichage de chaque ligne de code Perl au fur et à mesure de l'exécution, à la façon dont `sh -x` le fournit pour les scripts shell, vous ne pouvez pas utiliser l'option **-D** de Perl. Faites ceci à la place

```
SI vous disposez de l'utilitaire "env"
env PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

Syntaxe Bourne shell
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

Syntaxe csh
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

Voir *perldebug* pour plus de détails et des variantes.

### **-e** *lignedeprogramme*

peut être utilisé pour entrer une ligne de programme. Si **-e** est présent, Perl ne cherchera pas de nom de fichier dans la liste des arguments. Plusieurs options **-e** peuvent être combinées pour créer des scripts multi-lignes. Assurez-vous de bien mettre les points-virgules là où vous en mettriez dans un programme normal.

### **-f**

Désactive l'exécution de `$ Config{sitelib}/sitecustomize.pl` au démarrage.

Perl peut être compilé de manière à exécuter, par défaut, le script `$ Config{sitelib}/sitecustomize.pl` au démarrage. Ceci permet à l'administrateur de modifier le comportement de perl. Par exemple pour ajouter des répertoires au tableau `@INC` afin que perl trouve des modules à des emplacement non standard.

### **-Fmotif**

indique le motif à utiliser pour l'auto-découpage si **-a** est aussi présent. Le motif peut être délimité par `//`, `"` ou `"`, sinon il sera mis entre apostrophes.

### **-h**

affiche un résumé des options.

### **-i***[extension]*

indique que les fichiers traités par la forme `<>` doivent être édités sur place. Cela est accompli en renommant le fichier source, en ouvrant le fichier résultat sous le nom initial puis en sélectionnant ce fichier de résultat comme sortie par défaut pour les instructions `print()`. L'extension, si elle est fournie, est utilisée pour modifier le nom du fichier source pour faire une copie de sauvegarde, suivant ces règles :

Si aucune extension n'est fournie, aucune sauvegarde n'est faite et le fichier source est écrasé.

Si l'extension ne contient pas le caractère `*`, elle est ajoutée à la fin du nom de fichier courant comme suffixe. Si l'extension contient un ou plusieurs caractères `*`, chacune des `*` est remplacée par le nom de fichier courant. En perl, on pourrait l'écrire ainsi :

```
($backup = $extension) =~ s/*/$file_name/g;
```

Cela vous permet d'ajouter un préfixe au fichier de sauvegarde, au lieu (ou en plus) d'un suffixe :

```
$ perl -pi'orig_*' -e 's/bar/baz/' fileA # sauvegarde en 'orig_fileA'
```

Ou même de placer les sauvegardes des fichiers originaux dans un autre répertoire (à condition que ce répertoire existe déjà):

```
$ perl -pi'old/*.orig' -e 's/bar/baz/' fileA # sauvegarde en 'old/fileA.orig'
```

Ces exemples sont équivalents :

```
$ perl -pi -e 's/bar/baz/' fileA # écrase le fichier courant
$ perl -pi '*' -e 's/bar/baz/' fileA # écrase le fichier courant

$ perl -pi '.orig' -e 's/bar/baz/' fileA # sauvegarde en 'fileA.orig'
$ perl -pi '*.orig' -e 's/bar/baz/' fileA # sauvegarde en 'fileA.orig'
```

À partir du shell, écrire

```
$ perl -p -i.orig -e "s/foo/bar/; ..." "
```

revient au même qu'utiliser le programme :

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

qui est équivalent à :

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
 if ($ARGV ne $oldargv) {
 if ($extension !~ /*/) {
 $backup = $ARGV . $extension;
 }
 else {
 ($backup = $extension) =~ s/*/$ARGV/g;
 }
 rename($ARGV, $backup);
 open(ARGVOUT, ">$ARGV");
 select(ARGVOUT);
 $oldargv = $ARGV;
 }
 s/foo/bar/;
}
continue {
 print; # affiche le nom du fichier original
}
select(STDOUT);
```

si ce n'est que l'option **-i** ne compare pas \$ARGV et \$oldargv pour savoir quand le nom de fichier a changé. Cette option utilise par contre, ARGVOUT pour l'identificateur de fichier (filehandle). Notez que STDOUT est restauré comme sortie par défaut après la boucle.

Comme montré ci-dessus, Perl crée le fichier de sauvegarde même si la sortie n'est pas modifiée. C'est donc une manière amusante de copier des fichiers.

```
$ perl -p -i '/some/file/path/*' -e 1 file1 file2 file3...
```

ou

```
$ perl -p -i '.orig' -e 1 file1 file2 file3...
```

Vous pouvez utiliser eof sans parenthèses pour localiser la fin de chaque fichier d'entrée, au cas où vous voulez ajouter des choses à la fin ou réinitialiser le comptage des lignes (cf. exemples dans eof in *perlfunc*).

Si, pour un fichier donné, Perl n'est pas capable de créer de fichier de sauvegarde avec l'extension indiquée, il ne traitera pas le fichier et passera au suivant (s'il existe).

Pour une discussion des détaillée des permissions des fichiers et **-i**, voir Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi **-i** écrit dans des fichiers protégés ? N'est-ce pas un bug de Perl ? in *perlfqa5*

Vous ne pouvez pas utiliser **-i** pour créer des répertoires ou pour enlever des extensions à des fichiers.

Perl ne transforme pas les ~ dans les noms de fichiers, ce qui est une bonne chose, puisque certains l'utilisent pour leurs fichiers de sauvegarde :

```
$ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Remarquez que, puisque **-i** renomme ou efface le fichier original avant de créer un nouveau fichier du même nom, les liens hard d'Unix ne sont pas préservés.

Enfin, l'option **-i** n'empêche pas l'exécution si aucun fichier n'est fourni sur la ligne de commande. Dans ce cas, aucune sauvegarde n'est faite (puisque l'original ne peut être déterminé) et le traitement se fait de STDIN vers STDOUT, comme on peut s'y attendre.

### **-I**repertoire

Les répertoires spécifiés pas **-I** sont ajoutés en tête du chemin de recherche des modules (@INC) et indiquent au pré-processeur C où chercher les fichiers à inclure. Le pré-processeur C est appelé avec **-P** ; par défaut il cherche dans /usr/include et /usr/lib/perl.

### **-l**[octval]

permet le traitement automatique des fins de lignes. Cela a deux effets différents. Premièrement, \$/ (le séparateur d'enregistrements en entrée) est automatiquement enlevé (par chomp()) si utilisé en combinaison avec l'une des

options **-n** ou **-p**. Deuxièmement,  $\$ \backslash$  (le séparateur d'enregistrements en sortie) reçoit la valeur *octval* de telle manière que toutes les instructions `print` aient ce séparateur ajouté à la fin [du `print()`]. Si *octval* est omis,  $\$ \backslash$  prend la valeur de  $\$ /$ . Par exemple, pour limiter les lignes à 80 colonnes :

```
perl -lpe 'substr($_, 80) = ""'
```

Notez que l'affectation  $\$ \backslash = \$ /$  est faite quand l'option est rencontrée, donc le séparateur en entrée peut être différent du séparateur en sortie si l'option **-l** est suivie de l'option **-0** :

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

Cela affecte un retour-chariot `\n` à  $\$ \backslash$  et ensuite affecte le caractère nul (ASCII 0) à  $\$ /$ .

**-m[-]module**

**-M[-]module**

**-M[-]'module ...'**

**-[mM][-]module=arg[,arg]...**

**-mmodule** exécute `use module ()` ; avant d'exécuter votre programme.

**-Mmodule** exécute `use module` ; avant d'exécuter votre programme. Vous pouvez utiliser des apostrophes pour ajouter du code supplémentaire après le nom du module, par exemple, `'-Mmodule qw(toto titi)'`.

Si le premier caractère après **-M** ou **-m** est un moins (-), alors le 'use' est remplacé par 'no'.

Un peu de sucre syntaxique intégré permet d'écrire **-mmodule=toto,titi** ou **-Mmodule=toto,titi** comme un raccourci de `'-Mmodule qw(toto titi)'`. Cela évite de recourir à des apostrophes lorsqu'on importe des symboles. Le code généré par **-Mmodule=toto,titi** est `use module split(/,/ ,q{toto,titi})`. Notez que la forme = fait disparaître la distinction entre **-m** et **-M**.

Par conséquent **-Mfoo=number** n'effectuera jamais une vérification de version (à moins que la routine `foo::import()` la fasse elle-même ce qui peut arriver si `foo` hérite de `Exporter` par exemple).

**-n**

indique à Perl de considérer votre programme comme étant entouré par la boucle suivante, qui le fait itérer sur les noms de fichiers passés en arguments, à la manière de **sed -n** ou **awk** :

```
LINE:
while (<>) {
 ... # votre script va là
}
```

Notez que les lignes ne sont pas affichées par défaut. Cf. **-p** pour afficher les lignes traitées. Si un fichier passé en argument ne peut pas être ouvert, pour quelque raison que ce soit, Perl vous prévient, et passe au fichier suivant.

Voici un moyen efficace d'effacer tous les fichiers agés de plus d'une semaine :

```
find . -mtime +7 -print | perl -nle unlink
```

C'est plus rapide que d'utiliser l'option **-exec** de **find**, car vous ne lancez plus un processus pour chaque fichier à effacer. Cela souffre du bug entraînant une mauvaise gestion des fins de lignes dans les chemins, que vous pouvez régler si vous suivez l'exemple donné pour l'option **-0**.

Les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle des opérations, avant ou après la boucle implicite du programme, comme dans **awk**.

**-p**

indique à Perl de considérer votre programme comme étant entouré par la boucle suivante, qui le fait itérer sur les noms de fichiers passés en arguments, un peu à la manière de **sed** :

```
LINE:
while (<>) {
 ... # votre programme va là
} continue {
 print or die "-p destination : $!\n";
}
```

Si un fichier passé en argument ne peut pas être ouvert, pour quelque raison que ce soit, Perl vous prévient, et passe au fichier suivant. Notez que les lignes sont affichées automatiquement. Une erreur durant l'affichage est considérée comme fatale. Pour supprimer les affichages, utilisez l'option **-n**. L'option **-p** prend le dessus sur **-n**.

Les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle des opérations, avant ou après la boucle implicite, comme dans **awk**.

**-P <-P>**

**NOTE : l'usage de -P est très fortement déconseillé à cause des problèmes que cela pose, et en particulier une mauvaise portabilité.**

demande à Perl de faire passer votre programme dans le pré-processeur C avant la compilation. Étant donné que les commentaires Perl comme les directives de **cpp** commencent par le caractère #, il vaut mieux éviter de mettre des commentaires qui débutent par un mot réservé par le pré-processeur tels que "if", "else" ou "define".

Si vous envisagez d'utiliser -P, vous devriez aussi jeter un oeil au module `Filter::cpp` sur CPAN.

Voici une liste non-exhaustive des problèmes inhérents à -P :

- La ligne `#!` est supprimée, donc aucune option n'est prise en compte.
- Un `-P` dans la ligne `#!` ne fonctionne pas.
- **Toutes** les lignes qui commencent par un `#` (précédé éventuellement d'espaces) mais qui ne ressemblent pas à une commande `cpp` sont supprimées y compris celles dans les chaînes Perl, dans les expressions rationnelles et dans les here-docs.
- Sur certaines plateformes le pré-processeur C en sait trop : il reconnaît les commentaires à la C++, ceux qui commencent par `///  
//` et qui se terminent en fin de ligne. En conséquence, cela pose problème pour des constructions Perl courantes telle :  

```
s/foo//;
```

 qui, après -P, devienne du code illégal :  

```
s/foo
```

 Un moyen de contourner cela consiste à utiliser un autre séparateur que `/"` comme, par exemple, `!"` :  

```
s!foo! !;
```
- Outre un pré-processeur C fonctionnel, cela nécessite aussi un `sed` fonctionnel. Si vous n'êtes pas sur UNIX, vous risquez de ne pas avoir cette chance.
- Le numéro des lignes du script ne sont pas préservés.
- L'option `-x` ne fonctionne pas avec -P.

**-s**

active une analyse rudimentaire des arguments sur la ligne de commande situés après le nom du programme mais avant tout nom de fichier passé en argument (ou avant un `-`). Toute option trouvée là est retirée de `@ARGV` et définit la variable éponyme dans le programme Perl. Le programme suivant affiche "1" si et seulement si le programme est invoqué avec une option `-xyz`, et "abc" s'il est invoqué avec `-xyz=abc`.

```
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n"; }
```

Notez qu'un argument comme `-help` créera la variable `${-help}` qui n'est pas conforme avec `strict refs`. De plus, si vous utilisez cette option dans un script qui active les avertissements (`use warnings`), vous obtiendrez de nombreux avertissements "used only once" ("utilisé une seule fois").

**-S**

fait en sorte que Perl utilise la variable d'environnement `PATH` pour rechercher le programme (sauf si le nom du programme contient des séparateurs de répertoires).

Sur certaines plateformes, Perl ajoute des suffixes au nom du programme pendant sa recherche. Par exemple, sur les plateformes Win32, les suffixes `".bat"` et `".cmd"` sont ajoutés si la recherche du nom originel échoue et si le nom ne se termine pas déjà par l'un de ces suffixes. Si votre Perl a été compilé avec `DEBUGGING` activé, donner l'option `-Dp` à Perl montre la progression de la recherche.

Ceci est typiquement utilisé pour émuler le démarrage `#!` sur les plateformes qui ne le supportent pas. C'est aussi pratique pour déboguer un script qui utilise `#!` et qui doit donc être normalement trouvé par le mécanisme de recherche via `PATH` du shell.

Cet exemple fonctionne sur de nombreuses plateformes ayant un shell compatible avec le Bourne shell :

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
if $running_under_some_shell;
```

Le système ignore la première ligne et donne le programme à `/bin/sh`, qui essaye alors d'exécuter le programme Perl comme un script shell. Le shell exécute la deuxième ligne comme une commande normale, et ainsi démarre l'interpréteur Perl. Sur certains systèmes, `$0` ne contient pas toujours le chemin complet, l'option `-S` dit donc à Perl de rechercher le programme si nécessaire. Après que Perl ait localisé le programme, il analyse les lignes et les ignore car la variable `$running_under_some_shell` n'est jamais vraie. Si le programme doit être interprété par `csh`, vous devrez remplacer `${1+"$@"}` par `$*`, même si cela ne comprend pas les espaces (et les autres caractères équivalents) inclus dans la liste d'arguments. Pour démarrer `sh` plutôt que `csh`, certains systèmes peuvent nécessiter le remplacement de la ligne `#!` par une contenant juste un deux points, qui sera poliment ignoré par Perl. D'autres



systèmes ne peuvent pas contrôler cela, et ont besoin d'une construction totalement diabolique qui fonctionnera sous **cs**, **sh**, ou Perl, comme celle-ci :

```
eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
if $running_under_some_shell;
```

Si le nom de fichier fourni contient des séparateurs de répertoires (i.e. si c'est un chemin absolu ou relatif), et si le fichier n'est pas trouvé, les plateformes qui ajoutent des extensions de noms de fichiers le feront et essayeront de trouver le fichier avec ces extensions ajoutées, les unes après les autres.

Sur les plateformes compatibles DOS, si le programme ne contient pas de séparateurs de répertoires, il sera d'abord recherché dans le répertoire courant avant d'être recherché dans le PATH. Sur les plateformes Unix, le programme sera recherché strictement dans le PATH.

**-t**

Agit comme **-T** mais les vérifications de "souillure" n'engendrent que des avertissements au lieu d'une erreur fatale. Ces avertissements peuvent même être désactivés via `no warnings qw(taint)`.

**NOTE : ce n'est qu'un substitut de -T** qui ne devrait être utilisé que temporairement pour aider à la sécurisation d'un ancien code. Pour du code réellement utilisé en production ou pour du nouveau code écrit à partir de la feuille blanche, vous devriez toujours utiliser **-T**.

**-T**

force l'activation des vérifications de "souillure" (des données) pour que vous puissiez les tester. Ordinairement, ces vérifications sont faites seulement lorsqu'on exécute en `setuid` ou `setgid`. C'est une bonne idée de les activer explicitement pour les programmes exécutés à la demande de quelqu'un en qui vous n'avez pas une totale confiance, comme par exemple les programmes CGI ou tout serveur internet que vous pourriez écrire en Perl. Voir *perlsec* pour plus de détails. Pour des raisons de sécurité, cette option doit être vue par Perl très tôt ; cela signifie habituellement qu'elle doit apparaître le plus tôt possible dans la ligne de commande ou sur la ligne `#!` pour les systèmes qui le supportent.

**-u**

Cette option obsolète force Perl à écrire un `coredump` (une image mémoire) après la compilation de votre programme. Vous pouvez ensuite théoriquement prendre ce `coredump` et en faire un fichier exécutable en utilisant le programme **undump** (non fourni). Ceci accélère le démarrage au détriment de l'espace disque (que vous pouvez minimiser en strippant l'exécutable. Mais un exécutable "hello world" fait encore environ 200K sur ma machine). Si vous voulez exécuter une portion de votre programme avant le dump, utilisez l'opérateur `dump()` à la place. Note : la disponibilité de **undump** est spécifique à la plateforme et il peut donc ne pas être disponible pour un portage spécifique de Perl.

Cette option a été remplacée par le nouveau générateur de code du compilateur Perl. Voir *B* et *B::Bytecode* pour plus de détails.

**-U**

permet à Perl de réaliser des opérations non sûres. Actuellement, les seules opérations "non sûres" sont l'effacement des répertoires par `unlink` lors d'une exécution en tant que `superutilisateur`, et l'exécution de programmes `setuid` avec les vérifications de souillures fatales transformées en avertissements. Notez que l'option **-w** (ou la variable `$^W`) doit être utilisée en même temps que cette option pour effectivement *générer* les avertissements de vérification de souillure.

**-v**

affiche le numéro de version et le niveau de mise à jour (le niveau de patch) de votre exécutable perl.

**-V**

affiche un résumé des principales valeurs de configuration de perl et le contenu actuel de `@INC`.

**-V:nom**

affiche sur `STDOUT` la valeur de la variable de configuration dont le nom est fourni. Par exemple :

```
$ perl -V:libc
 libc='/lib/libc-2.2.4.so';
$ perl -V:lib.
 libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
 libc='/lib/libc-2.2.4.so';
$ perl -V:lib.*
 libpth='/usr/local/lib /lib /usr/lib';
 libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
 lib_ext='.a';
```

```
libc='/lib/libc-2.2.4.so';
libperl='libperl.a';
....
```

De plus, des deux-points supplémentaires permettent de contrôler le format d'affichage. Un deux-points final supprime les passages à la ligne et les ';' finaux, autorisant ainsi l'utilisation de cette option dans une ligne de commande du shell. (Moyen mnémotechnique : le séparateur de chemin ':'.)

```
$ echo "compression-vars: " 'perl -V:z.*: ' " are here !"
compression-vars: zcat='' zip='zip' are here !
```

Un deux-points supplémentaire en tête supprime le 'name=' de la réponse, autorisant ainsi l'ajout du nom que vous voulez. (Moyen mnémotechnique : étiquette vide.)

```
$ echo "goodvfork=" './perl -Ilib -V::usevfork'
goodvfork=false;
```

Les deux-points en tête et final peuvent être utilisés conjointement si vous voulez des valeurs sans nom. Remarques que dans le cas ci-dessous, les valeurs des variables de configuration de PERL\_API sont fournies dans l'ordre alphabétique (de leur nom).

```
$ echo building_on 'perl -V::osname: -V::PERL_API_.*:' now
building_on 'linux' '5' '1' '9' now
```

**-w**

affiche des avertissements concernant les constructions douteuses, telles que les noms de variables mentionnés une seule fois, les variables scalaires utilisées avant d'être définies, les sous-programmes redéfinis, les références aux handles de fichiers indéfinis ou aux handles de fichiers ouverts en lecture seule et sur lesquels vous essayez d'écrire, les valeurs utilisées comme des nombres et qui n'ont pas l'air d'être des nombres, les tableaux utilisés comme s'ils étaient des scalaires, les sous-programmes récursifs se rappelant eux-mêmes plus de 100 fois et d'innombrables autres choses.

Cette option ne fait que valider la variable interne `^$W`. Vous pouvez invalider certains avertissements ou les transformer en erreurs fatales de façon spécifique en cpatant les signaux `__WARN__`, comme décrit dans *perlvar* et dans *warn* in *perlfunc*. Voir aussi *perldiag* et *perltrap*. Une nouvelle façon de gérer finement les avertissements est aussi disponible si vous voulez en manipuler des classes entières ; voir *warnings* ou *perllexwarn*.

**-W**

valide tous les avertissements sans tenir compte de `no warnings` ou de `^$W`. Voir *perllexwarn*.

**-X**

invalide tous les avertissements sans tenir compte de `use warnings` ou de `^$W`. Voir *perllexwarn*.

**-x****-x répertoire**

indique à Perl que le programme est contenu dans un grand texte ACSII n'ayant rien à voir, comme par exemple un courrier électronique. Les déchets qui le précèdent seront ignorés jusqu'à la première ligne commençant par `#!` et contenant la chaîne "perl". Toute option significative sur cette ligne sera appliquée. Si un nom de répertoire est spécifié, Perl passera dans ce répertoire avant d'exécuter le programme. L'option **-x** contrôle seulement la destruction des déchets qui précèdent le script. Le programme doit se terminer par `__END__` si des déchets doivent être ignorés après lui (si on le désire, le programme peut traiter tout ou partie de ce qui le suit via le handle de fichier DATA).

## 34.3 ENVIRONNEMENT

**HOME**

Utilisée si `chdir` n'a pas d'argument.

**LOGDIR**

Utilisée si `chdir` n'a pas d'argument et si `HOME` n'est pas définie.

**PATH**

Utilisée lors de l'exécution de sous-processus, et dans la recherche du programme si **-S** est utilisée.

**PERL5LIB**

Une liste de répertoires dans lesquels on doit rechercher les fichiers de bibliothèques Perl avant de les rechercher dans la bibliothèque standard et le répertoire courant. Tous les sous-répertoires spécifiques à l'architecture courante sont automatiquement inclus s'ils existent aux endroits spécifiés. Si `PERL5LIB` n'est pas définie, `PERLLIB` est

utilisée. Les noms des répertoires sont séparés (comme pour PATH) par des deux-points sur les plateformes de type Unix et par des points-virgules sur Windows (le bon séparateur de répertoires est donné par la commande `perl -V:path_sep`).

Lors d'une exécution avec vérifications de souillure activée (soit parce que le programme est exécuté en `setuid` ou en `setgid`, soit parce que l'option `-T` est activée), aucune de ces deux variables n'est utilisée. Le script devrait dire à la place :

```
use lib "/my/directory";
```

### PERL5OPT

Options de ligne de commande. Les options dans cette variable sont considérées comme faisant partie de toute ligne de commande Perl. Seules les options `-[DIMUdmtw]` sont autorisées. Lors d'une exécution avec vérifications de souillure (si le programme est exécuté en `setuid` ou si `setgid`, ou si l'option `-T` est activée), cette variable est ignorée. Si `PERL5OPT` commence par `-T`, la vérification sera activée et toutes les options qui suivent seront ignorées.

### PERLIO

Une liste de filtres `PerlIO` utilisant l'espace ou le deux-points comme séparateur. Si `perl` est construit afin d'utiliser le système d'E/S `PerlIO` (c'est le cas par défaut) alors ces filtres sont utilisés pour les E/S de `perl`.

Par convention, on commence une suite de noms de filtres par deux-points (par exemple `:perlio`) pour mettre en évidence la similarité avec des "attributs" variables. Mais le code qui analyse cette chaîne de spécifications de filtres (et qui décode aussi la variable d'environnement `PERLIO`) considère les deux-points comme un séparateur.

Une variable `PERLIO` non définie ou vide est équivalent à `:stdio`.

Cette liste devient la valeur par défaut pour *toutes* les E/S de `perl`. Par conséquent, seuls les filtres déjà intégrés à `perl` peuvent apparaître dans cette liste. Les filtres externes (comme par exemple `:encoding()`) nécessitent des E/S utilisables pour pouvoir être chargés ! Voir "directive open" pour savoir comment ajouter des codages externes aux valeurs par défaut.

Voici une présentation rapide des filtres utilisables dans la variable d'encoding `PERLIO`. Pour plus de détails, consultez *PerlIO*.

#### :bytes

Un pseudo-filtre qui désactive le drapeau `:utf8` pour le filtre précédent. Rarement utilisable seul dans la variable d'environnement `PERLIO`. Peut-être intéressant dans des cas comme `:crlf:bytes` ou `:perlio:bytes`.

#### :crlf

Un filtre qui assure la traduction entre CRLF et `"\n"` en distinguant les fichiers "texte" et les fichiers "binaires" à la manière de MS-DOS et des systèmes d'exploitation assimilés. (Pour l'instant, il ne pousse pas le mimétisme jusqu'à considérer un Control-Z comme un marqueur de fin de fichier.)

#### :mmap

Un filtre qui implémente la "lecture" des fichiers via `mmap()` pour rendre accessible l'ensemble du fichier dans l'espace mémoire adressable du processus et pour l'utiliser comme "tampon" (buffer) `PerlIO`.

#### :perlio

C'est une ré-écriture sous forme de filtre `PerlIO` des méthodes de gestion des buffers à la "stdio". Pour toutes les opérations, il fait donc appel aux couches sous-jacentes (classiquement `:unix`).

#### :pop

Un pseudo-filtre expérimental qui permet de retirer le filtre le plus haut. À utiliser avec des pincettes, comme si c'était de la nitroglycérine.

#### :raw

Un pseudo-filtre qui manipule les autres filtres. L'application du filtre `:raw` est équivalent à un appel à `binmode($fh)`. Chaque octet du flux est passé tel quel sans aucune traduction. En particulier la traduction des CRLF ou les filtres `:utf8` déduits du locale courant sont désactivés.

Contrairement aux versions antérieures de Perl, `:raw` n'est *plus* l'exact inverse de `:crlf` - les autres filtres qui pourraient modifier la nature binaire du flux sont aussi supprimés ou désactivés.

#### :stdio

Ce filtre fournit une interface `PerlIO` pour accéder aux appels ANSI C de la couche "stdio" du système. Ce filtre fournit à la fois le tamponnage (la bufferisation) et les E/S. Notez que le filtre `:stdio` n'inclut pas la traduction du CRLF même si c'est le comportement normal sur la plateforme. Vous devez ajouter un filtre `:crlf` au-dessus pour le faire.

#### :unix

Filtre de bas niveau qui fait appel à `read`, `write`, `lseek`, etc.

**:utf8**

Un pseudo-filtre qui indique aux autres filtres que la sortie doit être en utf8 ou que les entrées doivent être considérées comme étant déjà sous la forme utf8. Peut-être utile dans la variable d'environnement PERLIO pour placer l'UTF-8 comme valeur par défaut. (Pour désactiver cela, utilisez le filtre `:bytes`.)

**:win32**

Sur les plateformes Win32, ce filtre *experimental* utilise les manipulateur d'E/S natif plutôt que les descripteurs numériques de fichier à la unix. Est reconnu comme étant bogué dans cette version.

Sur chacune des plateformes, l'ensemble de filtres choisis par défaut devrait donner des résultats corrects.

Pour les plateformes Unix, cela signifie "unix perlio" ou "stdio". Configure est réglé pour préférer l'implémentation "stdio" si la bibliothèque système propose des accès rapides aux tampons. Sinon, c'est "unix perlio" qui est proposé.

Sur Win32, le réglage par défaut de la distribution actuelle est "unix crlf". Le "stdio" de Win32 utilisé comme IO pour perl a trop de bugs ou de fonctionnalités bancales qui, de plus, dépendent parfois du fournisseur ou de la version du compilateur C utilisé. En utilisant notre propre filtre `crlf` pour la bufferisation permet d'éviter ces problèmes et rend les chose plus uniformes. Le filtre `crlf` fournit la conversion CRLF en "\n" et inversement, mais aussi les fonctionnalités de bufferisation (mise en mémoire tampon).

La distribution actuelle utilise `unix` comme second filtre sur Win32. Elle utilise donc les routines à base de descripteurs numériques de fichiers du compilateur C. C'est une couche native expérimentale pour `win32` qui devrait s'améliorer et deviendra un jour le choix par défaut pour Win32.

**PERLIO\_DEBUG**

Si cette variable contient le nom d'un fichier ou d'un périphérique alors certaines opérations du sous-système PerlIO seront tracées sur ce fichier (ouvert en ajout). Une utilisation classique sur Unix :

```
PERLIO_DEBUG=/dev/tty perl script ...
```

et sur Win32 :

```
set PERLIO_DEBUG=CON
perl script ...
```

Cette fonctionnalité n'est pas utilisable pour les script `setuid` ou ceux qui utilisent l'option `-T`.

**PERLLIB**

Une liste de répertoires ponctuée de deux points dans lesquels on doit rechercher les fichiers de bibliothèque de Perl avant de les rechercher dans la bibliothèque standard et le répertoire courant. Si `PERL5LIB` est définie, `PERLLIB` n'est pas utilisée.

**PERL5DB**

La commande utilisée pour charger le code du débogueur. La valeur par défaut est :

```
BEGIN { require 'perl5db.pl' }
```

**PERL5DB\_THREADED**

Si cette variable contient une valeur vraie, cela indique au débogueur que le code à déboguer utilise les fils d'exécution (les threads).

**PERL5SHELL (spécifique à la version WIN32)**

Peut être fixée vers un shell alternatif que perl doit utiliser en interne pour exécuter les commandes fournies entre accents graves ou par `system()`. La valeur par défaut est `cmd.exe /x/d/c` sous WindowsNT et `command.com /c` sous Windows95. La valeur est considérée comme délimitée par des espaces. Précédez tout caractère devant être protégé (comme un espace ou une barre oblique inverse) par une barre oblique inverse.

Notez que Perl n'utilise pas `COMSPEC` pour cela car `COMSPEC` a un haut degré de variabilité entre les utilisateurs, ce qui amène à des soucis de portabilité. De plus, perl peut utiliser un shell qui ne convienne pas à un usage interactif, et le fait de fixer `COMSPEC` vers un tel shell peut interférer avec le fonctionnement correct d'autres programmes (qui regardent habituellement `COMSPEC` pour trouver un shell permettant l'utilisation interactive).

**PERL\_ALLOW\_NON\_IFS\_LSP (spécifique à la version Win32)**

Si cette variable vaut 1, cela autorise l'utilisation de LSP non compatibles ISF. Perl essaye normalement d'utiliser des LSP compatibles ISF afin de permettre l'émulation de véritables filehandles au-dessus des sockets de Windows. Mais cela peut poser problème si vous utilisez un firewall tel que McAfee Guardian qui exige que toutes les applications utilisent ses propres LSP qui ne sont pas compatibles ISF. Perl ne le fera pas évidemment. En plaçant cette variable à 1, Perl utilisera le premier LSP utilisable dans le catalogue ce qui contentera McAfee Guardian (et, dans ce cas particulier, Perl continuera à fonctionner puisque les LSP de McAfee Guardian permettent, par des moyens détournés, le fonctionnement des applications exigeant la compatibilité IFS).

**PERL\_DEBUG\_MSTATS**

Valable uniquement si perl est compilé avec la fonction malloc incluse dans la distribution de perl (c.-à-d. si perl -V:d\_mymalloc vaut 'define'). Si elle a une valeur, cela provoque l'affichage de statistiques d'utilisation de la mémoire après l'exécution. Si sa valeur est un entier supérieur à un, les statistiques d'utilisation de la mémoire sont aussi affichées après la phase de compilation.

**PERL\_DESTRUCT\_LEVEL**

Valable uniquement si votre exécutable perl a été construit avec **-DDEBUGGING**, ceci contrôle le comportement de la destruction globale des objets et autres références. Voir PERL\_DESTRUCT\_LEVEL in *perlhack* pour plus d'informations.

**PERL\_DL\_NONLAZY**

Si cette variable vaut 1, perl tente de résoudre **tous** les symboles non-définis lors du chargement d'une bibliothèque dynamique. Le comportement par défaut est d'attendre l'utilisation d'un symbole pour effectuer sa résolution. En activant cette variable durant les tests d'une extension, vous détecterez les noms de fonctions mal orthographiés même si les test n'y font pas appel.

**PERL\_ENCODING**

Si vous utilisez la directive `encoding` sans fournir explicitement le nom du codage, la variable d'environnement PERL\_ENCODING est utilisé comme nom de codage.

**PERL\_HASH\_SEED**

(Existe depuis Perl 5.8.1.) Sa valeur est utilisée pour contrôler le caractère aléatoire de le fonction de hachage de Perl. Pour simuler le comportement des versions antérieures à la version 5.8.1, placez-y une valeur entière (avec la valeur zéro vous retrouverez le comportement exact de la version 5.8.0). Dans les versions antérieures à la 5.8.1, entre autres, l'ordre des clés des tables de hachage étaient toujours le même d'une exécution à l'autre de Perl.

Le comportement par défaut est de choisir un valeur aléatoire à moins que la variable PERL\_HASH\_SEED soit définie. Si Perl a été compilé avec l'option `-DUSE_HASH_SEED_EXPLICIT`, le comportement par défaut est de **ne pas** choisir une valeur aléatoire à moins que PERL\_HASH\_SEED soit définie.

Si PERL\_HASH\_SEED n'est pas définie ou ne contient pas une valeur numérique, Perl utilise une graine pseudo-aléatoire fournie par le système et ses bibliothèques. Cela signifie que chaque exécution de Perl proposera un ordre différent pour `keys()`, `values()` et `each()`.

**Notez bien que la valeur de cette graine est une information sensible.** L'ordre dans les tables de hachage est modifié aléatoirement pour protéger le code Perl d'attaque locale ou distante. En définissant manuellement cette graine, cette protection peut être partiellement ou complètement perdue.

Voir Attaques par complexité algorithmique in *perlsec* et PERL\_HASH\_SEED\_DEBUG pour plus d'information.

**PERL\_HASH\_SEED\_DEBUG**

(Existe depuis Perl 5.8.1) Placer cette variable d'environnement à un pour afficher (sur SDTERR) la valeur de la graine des tables de hachage au tout début de l'exécution. C'est prévu, en combinaison avec PERL\_HASH\_SEED pour aider au débogage malgré le comportement non-déterministe dû au caractère aléatoire de la fonction de hachage de Perl.

**Notez bien que la valeur de cette graine est une information sensible.** Sa connaissance permet de construire une attaque en déni de service contre Perl, même à distance. Voir Attaques par complexité algorithmique in *perlsec* et PERL\_HASH\_SEED\_DEBUG pour plus d'information. **Ne divulguez donc pas la valeur de cette graine** à ceux qui n'ont pas à la connaître. Voir aussi `hash_seed()` dans *Hash::Util*.

**PERL\_ROOT (spécifique au portage VMS)**

Un nom logique qui, sur VMS uniquement, traduit la racine cachée contenant perl et les chemins pour @INC. Parmi les autres noms logiques qui affectent perl sur VMS, on trouve PERLSHR, PERL\_ENV\_TABLES et SYS\$TIMEZONE\_DIFFERENTIAL mais ils sont optionnels et vous en saurez plus en lisant *perlvms* et le fichier *README.vms* dans la distribution des sources de Perl.

**PERL\_SIGNALS**

Dans Perl 5.8.1 et au-delà. Si cette variable contient `unsafe` (NdT : *non-sûr*), le comportement des versions pré-Perl-5.8.0 vis-à-vis de signaux est adopté (délivrance immédiate mais non sûre). Si cette variable contient `safe` (NdT : *sûr*), le comportement sûr (et différé) est utilisé. Voir Signaux différés (signaux sûrs) in *perlipc*.

**PERL\_UNICODE**

Équivalent à l'option `-C` de la ligne de commande. Notez que ce n'est pas une variable booléenne : y mettre la valeur "1" n'est pas le bon moyen pour "activer Unicode" (quelqu'en soit la signification). Par contre, vous pouvez "désactiver Unicode" en y plaçant la valeur "0" (ou en rendant PERL\_UNICODE indéfini dans votre shell avant de démarrer Perl). Voir la description de l'option `-C` pour plus d'information.

Perl utilise aussi de variables d'environnement pour contrôler la manière dont il manipule les données spécifiques à un langage naturel particulier. Voir *perllocale*.

À part toutes celles-ci, Perl n'utilise pas d'autres variables d'environnement, sauf pour les rendre accessibles au programme en cours d'exécution et à ses processus fils. Toutefois, les programmes exécutés en *setuid* feraient bien d'exécuter les lignes suivantes avant de faire quoi que ce soit d'autres, ne serait-ce que pour que les gens restent honnêtes :

```
$ENV{PATH} = '/bin:/usr/bin'; # ou ce que vous voulez
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

## 34.4 TRADUCTION

### 34.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 34.4.2 Traducteur

Pour la version 5.6.0 : Loic Tortay <[loict@bougon.net](mailto:loict@bougon.net)>, Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>. Pour la mise à jour en 5.8.8 : Paul Gaborit ([paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)).

### 34.4.3 Relecture

Personne pour l'instant.

# Chapitre 35

## perldiag

Les différents messages de Perl

### 35.1 DESCRIPTION

Les différents messages sont classés comme suit (listés dans l'ordre croissant d'importance) :

- (W) Avertissement (optionnel).
- (D) Désapprobation (optionnel).
- (S) Avertissement sérieux (impératif).
- (F) Erreur fatale capturable.
- (P) Erreur interne que vous ne verrez probablement jamais (capturable).
- (X) Erreur fatale non capturable
- (A) Erreur externe (non générée par Perl).

Les messages optionnels sont activés par l'option `-w`. Les avertissements peuvent être capturés en faisant pointer `$SIG{__WARN__}` vers une référence sur une fonction qui sera appelée à chaque avertissement au lieu d'être fiché. Voir *perlvar*. Les erreurs qui peuvent être capturées par l'opérateur `eval`. Voir `eval()` dans *perlfunc*.

Certains des messages sont génériques. Les endroits qui changent sont notés `%s`, comme dans le format de `printf`. Remarquez que certains messages commencent par `%s!` Les symboles `%(-?@` trient avant les lettres, alors que `[` et `\` trient après.

#### **"my" variable %s can't be in a package**

(F) La portée des variables n'est pas dans le paquetage, et cela n'a pas de sens d'essayer d'en déclarer une avec le nom du paquetage devant. Utiliser `local()` si vous voulez localiser une variable de paquetage.

#### **"my" variable %s masks earlier declaration in same scope**

(W) Une variable a été redéclarée dans le même bloc, ce qui élimine tous les accès à l'instance précédente. C'est le plus souvent une faute de frappe. Remarquez que la variable déclarée plus tôt continue d'exister en silence jusqu'à la fin du bloc ou jusqu'à ce que les références vers elle soient détruites.

#### **"no" not allowed in expression**

(F) Le mot clé « `no` » a été trouvé et exécuté à la compilation, et retourne une valeur non utilisable. Voir *perlmod*.

#### **"use" not allowed in expression**

(F) Le mot clé « `use` » a été trouvé et exécuté à la compilation et retourne une valeur non utilisable. Voir *perlmod*.

#### **% may only be used in unpack**

(F) Vous ne pouvez compacter une chaîne en soumettant un checksum, car le processus de checksum perd l'information, et vous ne pouvez aller dans l'autre sens. Voir `unpack()` dans *perlfunc*.

#### **%s (...) interpreted as function**

(W) Vous tombez dans la loi qui dit que chaque liste d'opérateurs suivie par des parenthèses est transformée en fonction, avec toute la liste d'arguments trouvés dans les parenthèses. Voir Termes et opérateurs de listes (leftward) in *perlop*.

#### **%s argument is not a HASH element**

(F) L'argument de `exists()` doit être un élément d'un tableau de hachage, comme

```
$foo{$bar}
$ref->[12]->{"susie"}
```

**%s argument is not a HASH element or slice**

(F) L'argument de `delete()` doit être ou un élément d'un tableau de hachage, comme

```
$foo{$bar}
$ref->[12]->{"susie"}
```

ou une partie d'un tableau de référence comme

```
@foo{$bar, $baz, $xyzyzy}
@{$ref->[12]}{"susie", "queue"}
```

**%s did not return a true value**

(F) Une librairie (ou un fichier utilisé) doit retourner une valeur vraie pour indiquer que la compilation est correcte et que l'initialisation de son code s'est bien passé. Il est habituel de placer « 1; » en fin de fichier, ou une quelconque valeur vraie. Voir `require()` dans *perlfunc*.

**%s found where operator expected**

(S) L'analyseur syntaxique de Perl sait si il attend un terme ou un opérateur. Si il voit ce qu'il sait être un terme alors qu'il s'attend à un opérateur, il donne ce message d'alerte. Habituellement, cela indique qu'un opérateur a été omis, comme une point virgule.

**%s had compilation errors**

(F) C'est le message final lorsque `perl -c` échoue.

**%s has too many errors**

(F) L'analyseur rend la main après 10 erreurs. Les messages suivants ne seraient pas sensés.

**%s matches null string many times**

(W) L'exemple que vous donnez entre dans une boucle infinie si le moteur d'expression rationnelle ne contrôle pas cela. Voir *perlre*.

**%s never introduced**

(S) Le symbole en question a été déclaré mais hors de portée avant qu'il soit possible de s'en servir.

**%s syntax OK**

(F) Message final lorsque `perl -c` réussit.

**%s: Command not found**

(A) Vous lancez accidentellement votre script par `csH` au lieu de Perl. Vérifiez la ligne avec `#!`, ou lancez votre script manuellement dans Perl.

**%s: Expression syntax**

(A) Vous lancez accidentellement votre script par `csH` au lieu de Perl. Vérifiez la ligne avec `#!`, ou lancez votre script manuellement dans Perl.

**%s: Undefined variable**

(A) Vous lancez accidentellement votre script par `csH` au lieu de Perl. Vérifiez la ligne avec `#!`, ou lancez votre script manuellement dans Perl.

**%s: not found**

(A) Vous lancez accidentellement votre script par le Bourne shell au lieu de Perl. Vérifiez la ligne avec `#!`, ou lancez votre script manuellement dans Perl.

**(Missing semicolon on previous line?)**

(S) C'est un message donne a titre indicatif en complément avec le message « %s found where operator expected ». Oublier un point virgule déclenche automatiquement ce message.

**-P not allowed for setuid/setgid script**

(F) Le script a voulu être ouvert par le préprocesseur C par son nom, ce qui entraîne une erreur d'exécution qui casse la sécurité.

**-T and -B not implemented on filehandles**

(F) Perl ne peut utiliser le tampon de sortie du descripteur de fichier quand il ne connaît pas le type de sortie. Vous devez utiliser un nom de fichier à la place.

**-p destination: %s**

(F) Une erreur est survenue sur la sortie standard invoquée par l'option `-p`. (Cette sortie est redirigée vers `STDOUT` à moins que vous la rediriez avec `select()`.)



**?+\* follows nothing in regexp**

(F) Vous commencez une expression rationnelle avec un quantifiant. Mettre un `'\'` si vous pensez au sens littéraire. Voir *perlre*.

**@ outside of string**

(F) Vous avez une enveloppe temporaire qui spécifie une position absolue en dehors de la chaîne à décompacter. Voir `pack()` dans *perlfunc*.

**accept() on closed fd**

(W) Vous tentez de faire un `accept()` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de `socket()`? Voir `accept()` dans *perlfunc*.

**Allocation too large: %lx**

(X) Vous ne pouvez allouer plus de 64K sur machine MS-DOS.

**Applying %s to %s will act on scalar(%s)**

(W) Les expressions de comparaison (`//`), substitution (`s///`), et translation (`tr///`) fonctionnent avec des valeurs scalaires. Si vous appliquez l'une d'elles sur un tableau ou un tableau de hachage, cela convertit le tableau en une valeur scalaire – la longueur du tableau ou les informations de peuplement du tableau de hachage – puis travaille sur la valeur scalaire. Ce n'est probablement pas ce que vous pensez faire. Voir `grep()` et `map()` dans *perlfunc* pour les alternatives.

**Arg too short for msgsnd**

(F) `msgsnd()` nécessite une chaîne au moins aussi longue que `sizeof(long)`.

**Ambiguous use of %s resolved as %s**

(W)(S) Vous exprimez quelque chose qui n'est pas interprété comme tel. Normalement, il est facile de clarifier la situation en ajoutant une quote manquante, un opérateur, une paire de parenthèses ou une déclaration.

**Ambiguous call resolved as CORE::%(s), qualify as such or use &**

(W) Une fonction que vous avez déclarée a le même nom qu'un mot-clef de Perl, et vous avez utilisé ce nom sans que l'on puisse faire de distinction entre l'un ou l'autre. Perl décide d'appeler la fonction interne car votre fonction n'est pas importée.

Pour forcer l'interprétation sur l'appel de votre fonction, soit vous mettez un ampersand avant le nom, soit vous appelez votre fonction précédée par son nom de paquetage. Alternativement, vous pouvez importer vos fonctions (ou prétendre le faire avec `use subs`).

Pour l'interpréter en silence comme un opérateur Perl, utilisez le préfixe `CORE::` sur l'opérateur (ex. `CORE::log($x)`) ou déclarer la fonction comme une méthode objet (voir *attrs*).

**Args must match #! line**

(F) L'émulateur `setuid` nécessite que les arguments évoqués correspondent avec ceux utilisé sur la ligne `#!`. Comme certains systèmes imposent un unique argument sur la ligne `#!`, essayez de combiner les opérateurs ; exemple, passez `-w -U` en `-wU`.

**Argument "%s" isn't numeric%s**

(W) La chaîne indiquée est utilisée comme un argument avec un opérateur qui s'attend à une valeur numérique. Si vous êtes chanceux, le message indique quel opérateur pose problème.

**Array @%s missing the @ in argument %d of %s()**

(D) Les versions vraiment antérieures de Perl permettaient d'omettre le `@` des tableaux à certains endroits. Ceci est maintenant largement obsolète.

**assertion botched: %s**

(P) Le paquetage `malloc` fourni avec Perl a subi une erreur interne.

**Assertion failed: file "%s"**

(P) Une affirmation générale a échoué. Le fichier en question doit être examiné.

**Assignment to both a list and a scalar**

(F) Si vous affectez à un opérateur conditionnel, les 2<sup>e</sup> et 3<sup>e</sup> arguments doivent ou être tous les deux des scalaires ou tous les deux des listes. Autrement, Perl ne connaît pas le contexte pour fournir le bon coté.

**Attempt to free non-arena SV: 0x%lx**

(P) Tous les objets SV sont supposés être alloués dans un espace qui sera nettoyé à la sortie du script. Un SV a été découvert en dehors de cet espace.

**Attempt to free nonexistent shared string**

(P) Perl maintient une table interne qui compte les références de chaînes pour optimiser le stockage et l'accès au tableau associatif et autres chaînes. Cela indique que quelqu'un essaye de décrémenter le compte de références d'un chaîne qui ne serait plus trouvée dans la table.

**Attempt to free temp prematurely**

(W) Les valeur « Mortalized » sont supposées être libérées par la fonction `free_tmpls()`. Cela indique que quelque chose d'autre a libéré le SV avant que la fonction `free_tmpls()` ait une chance de le faire, ce qui veut dire que la fonction `free_tmpls()` a libéré un scalaire non référencé au moment où vous essayez de le faire.

**Attempt to free unreferenced glob pointers**

(P) The reference counts got screwed up on symbol aliases.(???)

**Attempt to free unreferenced scalar**

(W) Perl va décrémenter une référence d'un compteur d'un scalaire pour voir si il arrive à 0, et découvre qu'il est déjà arrivé à 0 plus tôt, et qu'il se peut qu'il ait été libéré par `free()`, et en fait, il a probablement été libéré. Cela peut indiquer que `SvREFCNT_dec()` a été appelé trop de fois, ou que `SvREFCNT_inc()` a été appelé un trop petit nombre de fois, ou que le SV a été mortalized quand il n'aurait pas dû, ou que la mémoire a été corrompue.

**Attempt to pack pointer to temporary value**

(W) Vous essayez de passer une valeur temporelle (comme le résultat d'une fonction, ou d'une expression calculée) au « p » temporaire de `pack()`. Cela veut dire que le résultat contient un pointeur vers un endroit qui peut devenir invalide à tout moment, même avant la fin de l'expression en cours. Utilisez des valeurs littérales ou globales comme arguments du « p » temporaire de `pack` pour éviter ce message.

**Attempt to use reference as lvalue in substr**

(W) Vous soumettez une référence comme premier argument de `substr()` utilisé comme un lvalue, ce qui est vraiment étrange. Peut-être avez-vous oublié de la déréférencer en premier. Voir `substr()` dans *perlfunc*.

**Bad arg length for %s, is %d, should be %d**

(F) Vous passez un tampon de taille incorrecte à `msgctl()`, `semctl()` ou `shmctl()`. En C, la taille correcte est respectivement `sizeof(struct msqid_ds *)`, `sizeof(struct semid_ds *)`, et `sizeof(struct shmid_ds *)`.

**Bad filehandle: %s**

(F) Un symbole a été passé à quelque chose qui s'attend à un descripteur de fichier, mais le symbole n'a aucun descripteur associé. Peut-être n'avez-vous pas fait de `open()`, ou dans un autre paquetage.

**Bad free() ignored**

(S) Une routine interne a appelé `free()` sur quelque chose qui n'a jamais été `malloc()`é dans un premier temps. Obligatoire, mais peut être désactivé en positionnant la variable d'environnement `PERL_BADFREE` à 1.

Ce message peut être assez fréquent avec un fichier `DB_file` sur les systèmes avec les bibliothèques dynamiques en « dur », comme AIX et OS/2. C'est un bug de Berkeley DB qui n'est pas spécifié si vous utilisez la fonction système *forgiving* `malloc()`.

**Bad hash**

(P) Une des fonctions internes de hash a passé un pointeur HV null.

**Bad index while coercing array into hash**

(F) L'index a regardé dans le tableau associatif et a découvert que le 0<sup>ème</sup> élément du pseudo-tableau est illégal. Les valeurs d'index doivent être égales ou supérieures à 1. Voir *perlref*.

**Bad name after %s::**

(F) Vous démarrez un nom de symbole en utilisant un préfixe de paquetage, et vous n'avez pas fini le symbole. En particulier, vous ne pouvez pas interpoler les deux-points.

Donc

```
$var = 'myvar';
$sym = mypack::$var;
```

n'est pas la même chose que

```
$var = 'myvar';
$sym = "mypack::$var";
```

**Bad symbol for array**

(P) Une requête interne a voulu ajouter une entrée de tableau à quelque chose qui n'est pas un symbole d'entrée de tableau.

**Bad symbol for filehandle**

(P) Une requête interne a voulu ajouter un descripteur de fichier à quelque chose qui n'est pas un symbole d'entrée de tableau.

**Bad symbol for hash**

(P) Une requête interne a voulu ajouter un élément d'un tableau de hachage à quelque chose qui n'est pas un symbole d'entrée de tableau.

**Badly placed ()'s**

(A) Vous lancez accidentellement votre script par `cs` au lieu de Perl. Vérifiez la ligne avec `#!`, ou lancez votre script manuellement dans Perl.

**Bareword "%s" not allowed while "strict subs" in use**

(F) Avec "strict subs" en utilisation, un mot est seulement autorisé comme identifiant de fonction, dans la boucle en cours, ou à la gauche du symbole « => ». Peut-être devriez-vous déclarer au préalable votre sous-programme ?

**Bareword "%s" refers to nonexistent package**

(W) Vous utilisez un mot qualifiant de la forme `Foo : :`, mais le compilateur dit qu'il n'y pas d'autres utilisations de ce nom avant ce point. Peut-être devriez-vous déclarer au préalable votre paquetage ?

**BEGIN failed—compilation aborted**

(F) Une exception non capturable a été levée pendant l'exécution d'un sous-programme BEGIN. La compilation stoppe immédiatement et l'interpréteur s'arrête.

**BEGIN not safe after errors—compilation aborted**

(F) Perl a trouvé un sous-programme BEGIN {} (ou une directive use, qui implique un BEGIN {}) après qu'une ou plusieurs erreurs soient déjà survenues. Tant que l'environnement du BEGIN {} ne peut-être garanti (dû aux erreurs), et tant que le code qui suit dépend d'une opération correcte, Perl rend la main.

**bind() on closed fd**

(W) Vous essayez de faire un bind() sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de socket() ? Voir bind() dans *perlfunc*.

**Bizarre copy of %s in %s**

(P) Perl a détecté une tentative de copie d'une valeur interne qui n'est pas copiable.

**Callback called exit**

(F) Une fonction invoquée depuis un paquetage externe via `perl_call_sv()` s'est terminée en appelant `exit`.

**Can't "goto" outside a block**

(F) Un goto a été exécuté pour sauter ce qui semble être un bloc, excepté que ce n'est pas le bon bloc. Cela apparaît habituellement si vous essayez de sauter hors d'un bloc `sort()` ou d'une fonction, ce qui ne pas de sens. Voir à `goto()` dans *perlfunc*.

**Can't "goto" into the middle of a foreach loop**

(F) Un « goto » a été exécuté pour sauter au milieu d'une boucle `foreach`. Vous ne pouvez y aller depuis l'endroit où vous êtes. Voir `goto()` dans *perlfunc*.

**Can't "last" outside a block**

(F) L'état « last » a été exécuté pour sortir du bloc courant, excepté qu'il n'y a pas de bloc courant. Remarquez que les blocs « if » ou « else » ne comptent pas comme des blocs « loop », ou comme un bloc donné par `sort()`. Vous pouvez habituellement doubler les boucles pour obtenir le même effet, car la boucle intérieure est considérée comme un bloc qui boucle un seule fois. Voir `last()` dans *perlfunc*.

**Can't "next" outside a block**

(F) L'état « next » a été exécuté pour réitérer le bloc courant, mais il n'y a pas de bloc courant. Remarquez que les blocs « if » ou « else » ne comptent pas comme des blocs « loop », ou comme un bloc donné par `sort()`. Vous pouvez habituellement doubler les boucles pour obtenir le même effet, car la boucle intérieure est considérée comme un bloc qui boucle un seule fois. Voir `next()` dans *perlfunc*.

**Can't "redo" outside a block**

(F) L'état « redo » a été exécuté pour recommencer le bloc courant, mais il n'y a pas de bloc courant. Remarquez que les blocs « if » ou « else » ne comptent pas comme des blocs « loop », ou comme un bloc donné par `sort()`. Vous pouvez habituellement doubler les boucles pour obtenir le même effet, car la boucle intérieure est considérée comme un bloc qui boucle un seule fois. Voir `redo()` dans *perlfunc*.

**Can't bless non-reference value**

(F) Seules les références en dur peuvent être consacrées. C'est comme cela que Perl « renforce » l'encapsulation des objets. Voir *perlobj*.

**Can't break at that line**

(S) C'est un message d'erreur qui est seulement affiché pendant l'exécution avec le débogueur, indiquant que la ligne spécifiée n'est pas l'emplacement d'une expression qui peut être stoppée.

**Can't call method "%s" in empty package "%s"**

(F) Vous appelez une méthode correctement, et vous avez indiqué correctement le paquetage fonctionnant comme une classe, mais ce paquetage n'a RIEN de défini, et n'a pas de méthodes. Voir *perlobj*.

**Can't call method "%s" on unblest reference**

(F) Un appel de méthode doit savoir dans quel paquetage il est supposé être lancé. Cela se trouve habituellement dans la référence de l'objet que vous soumettez, mais vous ne soumettez pas de référence d'objet dans ce cas là. Une référence n'est pas une référence d'objet jusqu'à qu'il ait été consacré (blessed). Voir *perlobj*.

**Can't call method "%s" without a package or object reference**

(F) Vous utilisez la syntaxe d'un appel de méthode, mais l'endroit rempli par la référence d'objet ou le nom du paquetage contient une expression qui retourne une valeur définie qui n'est ni une référence d'objet ni un nom de paquetage. Quelque chose comme ça reproduit l'erreur :

```
$BADREF = 42;
process $BADREF 1, 2, 3;
$BADREF->process(1, 2, 3);
```

**Can't call method "%s" on an undefined value**

(F) Vous utilisez la syntaxe d'un appel de méthode, mais l'endroit rempli par la référence d'objet ou le nom du paquetage contient une valeur indéfinie. Quelque chose comme ça reproduit l'erreur :

```
$BADREF = undef;
process $BADREF 1, 2, 3;
$BADREF->process(1, 2, 3);
```

**Can't chdir to %s**

(F) Vous appelez `perl -x/foo/bar`, mais `/foo/bar` n'est pas un répertoire dans lequel vous pouvez entrer, probablement parce qu'il n'existe pas.

**Can't coerce %s to integer in %s**

(F) Certain types de SVs, en particulier la table des entrées des symboles réels (typeglobs), ne peuvent être forcés ou stoppés d'être ce qu'ils sont. Donc vous ne pouvez pas faire quelque chose comme ça :

```
*foo += 1;
```

Vous POUVEZ dire

```
$foo = *foo;
$foo += 1;
```

mais alors `$foo` ne contient plus de glob.

**Can't coerce %s to number in %s**

(F) Certain types de SVs, en particulier la table des entrées des symboles réels (typeglobs), ne peuvent être forcés ou stoppés d'être ce qu'ils sont.

**Can't coerce %s to string in %s**

(F) Certain types de SVs, en particulier la table des entrées des symboles réels (typeglobs), ne peuvent être forcés ou stoppés d'être ce qu'ils sont.

**Can't coerce array into hash**

(F) Vous utilisez un tableau là où un tableau associatif est attendu, mais le tableau n'a pas d'informations sur comment passer des clés aux indices de tableau. Vous ne pouvez faire cela qu'avec les tableaux dont leurs références associatives sont d'index 0.

**Can't create pipe mailbox**

(P) Une erreur spécifique à VMS. Le `process(???)` souffre de quotas dépassés ou d'autres problèmes de limitation.

**Can't declare %s in my**

(F) Seuls les scalaires, les tableaux et les tableaux associatifs peuvent être déclarés comme variables lexicales. Ils doivent avoir un identifiant ordinaire comme nom.

**Can't do inplace edit on %s: %s**

(S) La création du nouveau fichier a échoué a cause la raison indiquée.

**Can't do inplace edit without backup**

(F) Vous êtes sur un système comme MS-DOS qui s'embrouille si il essaie de lire un fichier supprimé (mais toujours ouvert). Vous devez dire `-i.bak`, ou quelque chose d'identique.

**Can't do inplace edit: %s > 14 characters**

(S) Il n'y a pas assez de place dans le nom de fichier pour faire un backup de ce fichier.

**Can't do inplace edit: %s is not a regular file**

(S) Vous essayez d'utiliser l'option `-i` sur un fichier spécial, comme un fichier de `/dev` ou un FIFO. Le fichier est ignoré.

**Can't do setegid!**

(P) L'appel à `setegid()` a échoué pour certaines raisons dans l'émulateur `setuid` de `suidperl`.

**Can't do seteuid!**

(P) L'émulateur `setuid` de `suidperl` a échoué pour certaines raisons.

**Can't do setuid**

(F) Cela veut typiquement dire que le perl ordinaire essaye d'exécuter `suidperl` pour faire une émulation `setuid`, mais ne peut le faire. Il cherche un nom de la forme `sperl5.000` dans le même répertoire que réside l'exécutable `perl`, typiquement sous `/usr/local/bin` sur les machines Unix. Si le fichier s'y trouve, vérifiez les permissions d'exécution. Si ce n'est pas cela, demandez à votre administrateur système pourquoi il l'a désactivé.

**Can't do waitpid with flags**

(F) Cette machine n'a pas ni `waitpid()` ni `wait4()`, donc seul `waitpid()` sans aucun paramètre est émulé (???).

**Can't do {n,m} with n > m**

(F) Le minimum doit être inférieur ou égal au maximum. Si vous voulez vraiment que votre expression rationnelle ne trouve rien, faites juste `{0}`. Voir *perlre*.

**Can't emulate -%s on #! line**

(F) La ligne `#!` spécifie une option qui n'a pas de sens à ce point. Par exemple, il serait idiot de mettre l'option `-x` sur cette ligne `#!`.

**Can't exec "%s": %s**

(W) L'appel à la fonction `system()`, `exec()`, ou un `open()` sur un pipe n'a pu s'exécuter pour la raison indiquée. Les raisons typiques sont : les permissions sur le fichier sont mauvaises, le fichier n'a pu être trouvé dans `$ENV{PATH}`, l'exécutable en question a été compilé sur une autre architecture, ou la ligne `#!` pointe vers un interpréteur qui ne peut pas être lancé pour une raison similaire. (Ou peut-être votre système ne supporte pas `#!` après tout).

**Can't exec %s**

(F) Perl essaye d'exécuter le programme indiqué car c'est ce qui est spécifié à la ligne `#!`. Si ce n'est pas ce que vous voulez, vous devrez mettre « perl » quelque part sur la première ligne.

**Can't execute %s**

(F) Vous utilisez l'option `-S`, mais la copie du script à exécuter trouvée dans le `PATH` n'a pas les bonnes permissions.

**Can't find %s on PATH, '' not in PATH**

(F) Vous utilisez l'option `-S`, mais le script à exécuter ne peut être trouvé dans le `PATH`, ou avec des permissions incorrectes. Le script existe dans le répertoire courant, mais `PATH` l'empêche de se lancer.

**Can't find %s on PATH**

(F) Vous utilisez l'option `-S`, mais le script à exécuter ne peut être trouvé dans le `PATH`.

**Can't find label %s**

(F) Vous dites par un « goto » d'aller à une étiquette qui n'est mentionnée nulle part, ou a un endroit qu'il n'est possible de joindre. Voir `goto()` dans *perlfunc*.

**Can't find string terminator %s anywhere before EOF**

(F) Les chaînes de caractères en Perl peuvent s'étirer sur plusieurs lignes. Ce message veut dire que vous avez oublié le délimitateur fermant l'expression. Comme les parenthèses protégées par des quotes comptent comme un niveau, il manque dans l'exemple suivant la parenthèse finale :

```
print q(The character '(' starts a side comment.);
```

Si vous obtenez cela depuis un « here-document », vous devez avoir mis des espaces blancs non visibles avant ou après la marque de fermeture. Un bon éditeur de programmeur doit pouvoir vous aider à les trouver.

**Can't fork**

(F) Une erreur fatale est survenue alors lors de l'essai d'appel à `fork` lorsqu'il a ouvert un pipe.

**Can't get filespec - stale stat buffer?**

(S) Une alerte spécifique à VMS. Cela est causé par les différences entre les contrôles d'accès sous VMS et sous le modèle Unix. Sous VMS, les contrôles d'accès sont faits par le nom de fichier, plutôt que par bits dans le tampon de stat, alors que les ACL et autres protections peuvent être pris dans le compte. Malheureusement, Perl suppose que le tampon de stat contient toutes les informations nécessaires et le fournit, au lieu de la spécification du fichier, à la fonction de contrôle d'accès. Il va essayer de retirer les spécifications du fichier en utilisant le nom de device et de

FID présent dans le tampon de stat, mais cela ne marchera que si vous avez fait un appel subséquent à la fonction CTRL stat(), car le nom de device est écrasé à chaque appel. Si cette alerte apparaît, la recherche de nom échoue, et la fonction de contrôle d'accès rend la main et retourne FALSE, juste pour être conservateur. (Remarque : la fonction de contrôle d'accès connaît l'opérateur stat de Perl et les tests de fichiers, donc vous ne devriez jamais voir ce message en réponse à une commande Perl ; cela arrive seulement si certains codes internes prennent les tampons de stat à la légère.)

#### Can't get pipe mailbox device name

(P) Une erreur spécifique à VMS. Après avoir créé une boîte aux lettres pour agir en tant que pipe, Perl ne peut retirer son nom pour un usage ultérieur.

#### Can't get SYSGEN parameter value for MAXBUF

(P) Une erreur spécifique à VMS. Perl a demandé à \$GETSYI quelle taille vous vouliez pour vos boîtes aux lettres, et n'a pas obtenu de réponse.

#### Can't goto subroutine outside a subroutine

(F) L'appel à la très magique « goto subroutine » peut seulement remplacer l'appel d'une fonction par une autre. Il ne peut fabriquer one out of whole cloth(???). En général, elle ne peut être appelé que depuis une fonction d'AUTOLOAD. Voir goto() dans *perlfunc*.

#### Can't goto subroutine from an eval-string

(F) L'appel à « goto subroutine » ne peut être utilisé pour sortir d'une chaîne utilisée par eval(). (Vous pouvez l'utiliser pour sortir d'un eval() {BLOCK}, mais ce n'est sûrement pas ce que vous voulez.

#### Can't localize through a reference

(F) Vous dites quelque chose comme local \$\$ref, ce que Perl ne peut accepter, car quand il va restaurer l'ancienne valeur de ce que \$ref référençait, il ne peut être sûr que \$ref est toujours une référence.

#### Can't localize lexical variable %s

(F) Vous utilisez local sur une variable qui a déjà été déclarée auparavant comme une variable lexicale à l'aide de « my ». Cela n'est pas permis. Si vous voulez localiser une variable de paquetage du même nom, qualifiez-la avec le nom du paquetage.

#### Can't localize pseudo-hash element

(F) Vous dites quelque chose comme local \$ar->{'key'}, où \$ar est une référence vers un pseudo-hash. Cela n'a pas été implémenté pour le moment, mais vous pouvez obtenir un effet similaire en localisant l'élément de tableau correspondant par - local \$ar->[\$ar->[0]{'key'}].

#### Can't locate auto/%s.al in @INC

(F) Une fonction (ou une méthode) a été appelé dans un paquetage qui autorise le chargement automatique, mais il n'y a pas de fonction à charger automatiquement. Le plus probable est une faute de frappe sur le nom de fonction/méthode ou une erreur pour utiliser AutoSplit sur le fichier, entraîné par make install.

#### Can't locate %s in @INC

(F) Vous dites d'exécuter (run) (ou require, ou use) un fichier qui ne peut être trouvé dans aucune des bibliothèques mentionnées dans @INC. Peut-être devriez-vous positionner la variable d'environnement PERL5LIB ou PERL5OPT pour dire où se trouve votre bibliothèque supplémentaire, ou peut-être le script nécessite que vous ajoutiez le nom de votre bibliothèque à @INC. Ou peut-être avez-vous mal épilé le nom de votre fichier. Voir require() dans *perlfunc*.

#### Can't locate object method "%s" via package "%s"

(F) Vous appelez un méthode correctement, et vous avez indiqué correctement un paquetage fonctionnant comme une classe, mais le paquetage ne définit pas cette méthode, ni aucune de ses classes parentes. Voir *perlobj*.

#### Can't locate package %s for @%s::ISA

(W) Le tableau @ISA contient un nom d'un autre paquetage qu'il ne semble pas exister.

#### Can't make list assignment to \%ENV on this system

(F) L'affectation de listes sur %ENV n'est pas supportée sur certains systèmes, notamment VMS.

#### Can't modify %s in %s

(F) Vous n'êtes pas autorisé à faire d'affectation sur l'élément indiqué, ou autrement essayez de le changer, comme avec une incrémentation automatique.

#### Can't modify nonexistent substring

(P) La fonction interne qui fait l'affectation à substr() a capturé un NULL.

#### Can't msgrcv to read-only var

(F) La cible de msgrcv doit être modifiable pour être utilisée comme un buffer reçu.

**Can't open %s: %s**

(S) L'ouverture implicite d'un fichier en utilisant `<>` sur le descripteur de fichier, implicitement via les options `-n` ou `-p` en ligne de commandes, ou explicitement, a échoué à cause de la raison indiquée. Habituellement c'est parce que vous n'avez pas les permissions de lecture sur le fichier que vous avez nommé sur la ligne de commande.

**Can't open bidirectional pipe**

(W) Vous essayez de dire `open(CMD, "|cmd|")`, ce qui n'est pas supporté. Vous pouvez essayer un des nombreux modules de la librairie Perl pour faire cela, comme `IPC::Open2`. Alternativement, dirigez la sortie du pipe dans un fichier en utilisant `<>`, et ensuite lisez-le dans un descripteur de fichier différent.

**Can't open error file %s as stderr**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commande, et ne peut ouvrir le fichier spécifié après `'2>'` ou `'2>>'` sur la ligne de commande pour y écrire.

**Can't open input file %s as stdin**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commande, et ne peut ouvrir le fichier spécifié après `'<'` sur la ligne de commande pour le lire.

**Can't open output file %s as stdout**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commande, et ne peut ouvrir le fichier spécifié après `'>'` ou `'>>'` sur la ligne de commande pour y écrire.

**Can't open output pipe (name: %s)**

(P) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commande, et ne peut ouvrir le pipe destiné à la sortie des données.

**Can't open perl script "%s": %s**

(F) Le script spécifié ne peut être ouvert pour la raison indiquée.

**Can't redefine active sort subroutine %s**

(F) Perl optimise l'appel interne aux sous-programmes `sort` et garde des pointeurs vers ceux-ci. Vous essayez de redéfinir un des sous-programmes qui est actuellement actif, ce qui n'est pas permis. Si vous voulez vraiment faire cela, vous pouvez écrire `sort { &func } @x` au lieu de `sort func @x`.

**Can't rename %s to %s: %s, skipping file**

(S) L'opération pour renommé effectué par l'option `-i` a échoué pour une raison, probablement parce que vous n'avez pas le droit d'écrire dans le répertoire.

**Can't reopen input pipe (name: %s) in binary mode**

(P) Une erreur spécifique à VMS. Perl pense que `stdin` est un pipe, et essaye de le rouvrir pour le faire accepter des données. Hélas, cela échoue.

**Can't reswap uid and euid**

(P) L'appel à la fonction `setreuid()` a échoué pour un raison dans l'émulateur `setuid` de `suidperl`.

**Can't return outside a subroutine**

(F) L'expression `return` a été exécutée dans la ligne principale, où il n'y a aucun appel de sous-programme à retourner. Voir *perlsub*.

**Can't stat script "%s"**

(P) Pour certaines raisons, vous ne pouvez faire un `fstat()` sur le script tant que vous l'avez déjà ouvert. Bizarre.

**Can't swap uid and euid**

(P) L'appel à la fonction `setreuid()` a échoué pour un raison dans l'émulateur `setuid` de `suidperl`.

**Can't take log of %g**

(F) Pour les nombres réels ordinaires, vous ne pouvez faire le logarithme d'un nombre négatif ou nul. Il existe un paquetage `Math::Complex` en standard avec Perl, si c'est vraiment pour faire cela sur un nombre négatif.

**Can't take sqrt of %g**

(F) Pour les nombres réels ordinaires, vous ne pouvez obtenir la racine carrée d'un nombre négatif. Il existe le paquetage `Math::Complex` en standard avec Perl, qui lui, si vous le voulez vraiment, peut faire cela.

**Can't undef active subroutine**

(F) Vous ne pouvez effacer une fonction qui est utilisée actuellement.

Vous pouvez cependant, la redéfinir pendant qu'elle tourne, et faire un `undef` sur la fonction redéfinie pendant que l'ancienne fonction tourne. Go figure(???)

**Can't unshift**

(F) Vous essayez de faire un `unshift` sur un tableau « irréal » qui ne peut accepter cette opération, comme la pile principale de Perl.

**Can't upgrade that kind of scalar**

(P) La fonction interne `sv_upgrade` a ajouté un « membre » à un SV, ce qui fait qu'il se trouve dans une sorte de SV plus spécialisée. Les différents premiers types de SV sont si spécialisés, cependant, qu'ils ne peuvent être inter-convertis. Ce message indique qu'une telle conversion a été tentée.

**Can't upgrade to undef**

(P) La non définie SV est le bas du « totem pole », dans le projet des « upgradability ». Arriver à undef indique une(???) dans le code qui appelle `sv_upgrade`.

**Can't use %%! because Errno.pm is not available**

(F) La première fois que le tableau associatif %! est utilisé, perl automatiquement charge le module `Errno.pm`. Le module `Errno` est attendu pour attacher au tableau %! les noms symboliques pour les valeurs des numéros d'erreur de \$!.

**Can't use "my %s" in sort comparison**

(F) Les variables globales \$a et \$b sont réservés pour les comparaisons avec `sort`. Vous mentionnez \$a ou \$b dans la même ligne que l'opérateur `<=>` ou `cmp`, et la variable qui a été déclarée plus tôt comme variable lexicale. Ou qualifiez la variable de `sort` avec le nom de paquetage, ou renommez la variable lexicale.

**Can't use %s for loop variable**

(F) Seulement une variable scalaire simple peut être utilisée comme variable de boucle dans un `foreach`.

**Can't use %s ref as %s ref**

(F) Vous avez mélangé vos types de références. Vous avez déréféréncé la référence du type nécessaire. Vous pouvez utiliser la fonction `ref()` pour tester le type de la référence, si nécessaire.

**Can't use \1 to mean \$ 1 in expression**

(W) Dans les expressions ordinaires, la barre oblique inverse est un opérateur unaire qui crée une référence vers cet argument. L'utilisation de barre oblique inverse pour indiquer une backreference sur une sous-chaîne correspondante est valide seulement comme partie du motif de l'expression rationnelle. Essayer de faire ça dans un code ordinaire Perl produit une valeur qui ressemble à `SCALAR(0xdecacaf)`. Utilisez l'élément \$1 à la place.

**Can't use bareword ("%s") as %s ref while "strict refs" in use**

(F) Les références en dur sont les seules autorisées par « strict refs ». Les références symboliques sont interdites. Voir *perlref*.

**Can't use string ("%s") as %s ref while "strict refs" in use**

(F) Seules les affectations « en dur » sont permises par « strict refs ». Les références symboliques sont interdites. Voir *perlref*.

**Can't use an undefined value as %s reference**

(F) Une valeur utilisée comme une référence en dur ou comme une référence symbolique doit avoir une valeur définie. Cela aide à débloquer certaines erreurs insidieuses.

**Can't use global %s in "my"**

(F) Vous essayez de déclarer une variable magique en tant que variable lexicale. Cela n'est pas permis, car la magique peut être lié qu'à un seul endroit (nommé variable globale) et il serait incroyablement confus d'avoir des variables dans votre programme qui ressemblent à une variable magique mais qui n'en sont pas une.

**Can't use subscript on %s**

(F) Le compilateur essaye d'interpréter une expression entre parenthèses comme un sous-script. Mais à gauche des parenthèses il y a une expression qui ne ressemble pas à un tableau de références, ni quoi que ce soit que l'on puisse interpréter comme un sous-script.

**Can't x= to read-only value**

(F) Vous essayez de répéter une valeur constante (souvent une valeur indéfinie) avec un opérateur d'affectation, ce qui implique de modifier la valeur elle-même. Peut-être que vous devriez copier la valeur dans un endroit temporaire, et recommencer.

**Cannot find an opnumber for "%s"**

(F) Une chaîne de la forme `CORE:::word` a été donné à `prototype()`, mais il n'existe pas de variable interne nommée `word`.

**Cannot resolve method '%s' overloading '%s' in package '%s'**

(F|P) Une erreur lors de la résolution de la surcharge spécifiée par le nom de méthode (à l'opposé d'une référence de sous-fonction) : aucune méthode de ce nom là peut être appelé via ce paquetage. Si le nom de la méthode est ???, il s'agit d'une erreur interne.



**Character class syntax [. ] is reserved for future extensions**

(W) À l'intérieur des classes de caractères dans les expressions rationnelles ([ ]) la syntaxe commençant par « [. » et se terminant par « .] » est réservé pour les extensions futures. Si vous devez représenter cette séquence de caractères dans une classe de caractères dans une expression rationnelle, cotez simplement les crochets avec une barre oblique inverse : « \[. » et « \.] ».

**Character class syntax [: ] is reserved for future extensions**

(W) À l'intérieur des classes de caractères dans les expressions rationnelles ([ ]) la syntaxe commençant par « [: » et se terminant par « :] » est réservé pour les extensions futures. Si vous devez représenter cette séquence de caractères dans une classe de caractères dans une expression rationnelle, cotez simplement les crochets avec une barre oblique inverse : « \[: » et « \:] ».

**Character class syntax [= ] is reserved for future extensions**

(W) À l'intérieur des classes de caractères dans les expressions rationnelles ([ ]) la syntaxe commençant par « [= » et se terminant par « =] » est réservé pour les extensions futures. Si vous devez représenter cette séquence de caractères dans une classe de caractères dans une expression rationnelle, cotez simplement les crochets avec une barre oblique inverse : « \[= » et « =\] ».

**chmod: mode argument is missing initial**

(W) Un utilisateur novice aura tendance à faire

```
chmod 777, $filename
```

en ne réalisant pas que `777` est interprété comme un nombre décimal, équivalent à `01411`. Les constantes octales sont introduites en utilisant un `0` en Perl, comme en C.

**Close on unopened file <%s>**

(W) Vous essayez de fermer un descripteur de fichier qui n'a jamais été ouvert.

**Compilation failed in require**

(F) Perl ne peut compiler le fichier spécifié à l'aide la directive `require`. Perl utilise ce message générique quand aucune des erreurs rencontrées n'est assez grave pour stopper la compilation immédiatement.

**Complex regular subexpression recursion limit (%d) exceeded**

(W) Le moteur d'expressions rationnelles utilise la récursion dans les situations complexe où le back-tracking est nécessaire. La profondeur de la récursion est limitée à 32766, ou peut-être moins sur les architectures où la pile ne peut grossir arbitrairement. (Les situations « simples » et « moyennes » peuvent être capturées sans récursion et ne sont pas sujet à une limite.) Essayer de raccourcir la chaîne examinée, et boucler sur le code Perl (ex : avec `while` plutôt que par le moteur d'expressions rationnelles ; ou réécrivez votre expression rationnelle pour qu'elle soit plus simple ou moins référencée. (Voir *perlbook* pour des informations sur *Mastering Regular Expressions*.)

**connect() on closed fd**

(W) Vous essayez de faire un `connect` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel à `socket()` ? Voir `connect()` dans *perlfunc*.

**Constant subroutine %s redefined**

(S) Vous redéfinissez une fonction qui a été marquée auparavant comme inlining. Voir Fonctions Constantes in *perlsub* pour les commentaires et les issues.

**Constant subroutine %s undefined**

(S) Vous supprimez une fonction qui a été marquée auparavant comme inlining. Voir Constant Functions in *perlsub* pour les commentaires et les issues.

**Copy method did not return a reference**

(F) La méthode qui surclasse « = » est buggée. Voir Copy Constructor in *overload*.

**Corrupt malloc ptr 0x%lx at 0x%lx**

(P) Le paquetage `malloc` distribué avec Perl a eu une erreur interne.

**corrupted regexp pointers**

(P) Le moteur d'expressions rationnelles a été abusé par l'expression rationnelle que vous lui avez fournie.

**corrupted regexp program**

(P) Le moteur d'expressions rationnelles a analysé une regexp sans un « magic number » valide.

**Deep recursion on subroutine "%s"**

(W) Ce sous-programme s'est appelé lui-même (directement ou indirectement) 100 fois plus de fois qu'il n'a retourné une valeur. Cela indique probablement une récursion infinie, à moins que vous n'écriviez un étrange programme de benchmark, dans ce cas cela indique quelque chose d'autre.

**Delimiter for here document is too long**

(F) Dans un « here document » construit comme <<FOO, l'étiquette FOO est trop long pour que Perl puisse la traiter. Vous devez être sérieusement tordu pour écrire un code qui entraîne cette erreur.

**Did you mean & %s instead?**

(W) Vous faites probablement référence à une sous-fonction importée &FOO en faisant \$FOO ou quelque chose du genre.

**Did you mean \$ or @ instead of %?**

(W) Vous avez probablement dit %hash{\$key} alors que vous pensiez \$hash{\$key} ou @hash{@keys}. En d'autres termes, peut-être pensez-vous à %hash et vous avez et ca l'a emporté.???

**Died**

(F) Vous appelez die() avec une chaîne vide (l'équivalent de die "") ou vous l'appelez sans arguments et avec @\$ et \$\_ vides.

**Do you need to predeclare %s?**

(S) C'est un message donné à titre indicatif en conjonction avec le message « %s found where operator expected ». Cela veut souvent dire qu'un nom de fonction ou de module est référencé alors qu'il n'est pas encore défini pour le moment. C'est peut être un problème d'ordre dans votre fichier, ou parce qu'il manque un « sub », « paquetage », « require », ou « use ». Si vous référencez quelque chose qui n'est pas encore défini pour le moment, vous n'avez pas à définir la fonction ou le paquetage avant cet endroit. Vous pouvez utiliser « un sub foo; » ou un paquetage « FOO; » vide pour entrer une déclaration « devant ».

**Don't know how to handle magic of type '%s'**

(P) Le traitement interne des variables magiques a été endommagé.

**do\_study: out of memory**

(P) Ce message peut être capturé par l'appel de safemalloc().

**Duplicate free() ignored**

(S) Une fonction interne a appelé free() sur quelque chose qui a déjà été libéré.

**elsif should be elsif**

(S) Il n'y a pas de mot-clef « elsif » en Perl car Larry pense que c'est très laid. Votre code sera interprété comme un essai d'appel à la méthode nommée « elsif » pour la classe retournée par le bloc suivant. Ce n'est sûrement pas ce que vous voulez.

**END failed—cleanup aborted**

(F) Une exception non capturable a été levée pendant que la sous-fonction END était exécuté. L'interpréteur est sorti immédiatement.

**Error converting file specification %s**

(F) Une erreur spécifique à VMS. Comme Perl doit traiter avec des spécifications de fichiers autres que la syntaxe VMS ou Unix, il convertit ceux-ci dans un format unique où il peut opérer avec directement. Ou vous avez passé des spécifications de fichier non valide, ou vous avez trouvé un cas où la routine de conversion ne peut rien faire. Drat.???

**%s: Eval-group in insecure regular expression**

(F) Perl a détecté des données souillées quand il essaye de compiler une expression rationnelle qui contient le (?{ ... }) d'affectation de longueur zéro, ce qui n'est pas sécurisé. Voir (?{ code }) in *perlre*, and *perlsec*.

**%s: Eval-group not allowed, use re 'eval'**

(F) Une expression rationnelle contient une longueur nulle dans (?{ ... }), mais cette construction est seulement permise quand le code use re 'eval' est en action. Voir (?{ code }) in *perlre*.

**%s: Eval-group not allowed at run time**

(F) Perl essaye de compiler une expression rationnelle qui contient (?{ ... }) affectation de longueur nulle à l'exécution, comme si l'expression contenait les valeurs interpolées. Comme c'est un risque de sécurité, cela n'est pas permis. Si vous insistez, vous devez le faire en construisant explicitement votre expression depuis une chaîne interpolée à l'exécution et l'utiliser dans un eval().

**Excessively long <> operator**

(F) Le contenu des opérateurs <> ne doit pas excéder la taille maximum d'un identifiant Perl. Si vous essayez juste d'obtenir les extensions d'une longue liste de fichiers, essayez d'utiliser l'opérateur glob(), ou mettez le noms de fichiers dans une variable et faites un glob() dessus.

**Execution of %s aborted due to compilation errors**

(F) Le message final lorsqu'une compilation Perl échoue.

**Exiting eval via %s**

(W) Vous êtes sorti de Perl d'une façon non conventionnelle, comme un goto, ou un contrôle de boucle.

**Exiting pseudo-block via %s**

(W) Vous êtes sorti d'un bloc spécial de constructeur (comme un bloc sort ou une fonction) d'une façon non conventionnelle, comme un goto, ou un contrôle de boucle. Voir `sort()` dans *perlfunc*.

**Exiting subroutine via %s**

(W) Vous êtes sorti d'une fonction d'une façon non conventionnelle, comme un goto, ou un contrôle de boucle.

**Exiting substitution via %s**

(W) Vous êtes sorti d'une substitution d'une façon non conventionnelle, comme un goto, un return, ou un contrôle de boucle.

**Explicit blessing to '' (assuming package main)**

(W) Vous consacrez (to bless) une référence à une chaîne de longueur nulle. Cela a pour effet de consacrer la référence dans le paquetage principal. Ce n'est pas habituellement ce que vous voulez. Fournissez un paquetage-cible par défaut, ex `bless($ref, $p or 'MyPaquetage')`;

**Fatal VMS error at %s, line %d**

(P) Erreur spécifique à VMS. Il s'est passé quelque chose dans un service du système VMS ou d'une fonction RTL ; L'état de Perl à sa sortie peut fournir plus de détails. Le nom de fichier dans « at %s » et le numéro de ligne dans « line %d » vous indique quelle section du code source Perl est gêné.

**fcntl is not implemented**

(F) Votre machine apparemment n'implémente pas `fcntl()`. C'est quoi, un PDP-11 ou quelque chose de similaire ?

**Filehandle %s never opened**

(W) Une opération d'E/S a été tentée sur un descripteur de fichier qui n'a jamais été initialisé. Vous devez faire un appel à `open()` ou à `socket()`, ou appelez un constructeur depuis le paquetage `FileHandle`.

**Filehandle %s opened for only input**

(W) Vous tentez d'écrire dans un descripteur de fichier ouvert en lecture seulement. Si vous désirez que ce soit un descripteur de fichier ouvert en écriture, vous devez l'ouvrir avec « +< » ou « \$+> » ou « +>> » au lieu de « < » ou rien du tout. Si vous désirez uniquement écrire dans le fichier, utilisez « > » ou ">>". Voir `open()` dans *perlfunc*.

**Filehandle opened for only input**

(W) Vous tentez d'écrire dans un descripteur de fichier ouvert en lecture seulement. Si vous désirez que ce soit un descripteur de fichier ouvert en écriture, vous devez l'ouvrir avec « +< » ou « +> » ou « +>> » au lieu de « < » ou rien du tout. Si vous désirez uniquement écrire dans le fichier, utilisez « > » ou ">>". Voir `open()` dans *perlfunc*.

**Final \$ should be \\$ or \$ name**

(F) Vous devez maintenant décider si le \$ final dans une chaîne doit être interprétée comme le signe littéraire dollar, ou doit être interprétée comme l'introduction d'un nom de variable qui apparaît comme manquant. Donc vous devez ou mettre la barre oblique inverse ou bien le nom.

**Final @ should be \@ or @name**

(F) Vous devez maintenant décider si le @ dans une chaîne doit être interprété comme le signe littéral « at », ou si c'est dans le but d'introduire un nom de variable qui apparaît comme manquant. Donc vous devez mettre soit une barre oblique inverse soit le nom.

**Format %s redefined**

(W) Vous redéfinissez un format. Pour supprimer ce message faire

```
{
 local $^W = 0;
 eval "format NAME = ...";
}
```

**Format not terminated**

(F) Un format doit être terminé par une ligne avec un point uniquement. Perl est arrivé en fin de fichier sans trouver une telle ligne.

**Found = in conditional, should be ==**

(W) Vous dites

```
if ($foo = 123)
```

alors que vous pensez

```
if ($foo == 123)
```

(ou quelque chose de similaire).

**gdbm store returned %d, errno %d, key "%s"**

(S) Un message d'alerte de l'extension GDBM\_File qui a échoué un stockage.

**gethostent not implemented**

(F) Votre librairie C n'implémente apparemment pas `gethostent()`, probablement parce que si elle le fait, il se sentirait moralement obligé de rendre chaque hostname d'Internet.

**get{sock,peer}name() on closed fd**

(W) Vous essayez d'obtenir une socket ou une socket parente d'une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel à `socket()` ?

**getpwnam returned invalid UIC %#o for user "%s"**

(S) Message spécifique à VMS. L'appel à `sys$getuai` souligne l'opérateur `getpwnam` qui a retourné une UIC non valide.

**Glob not terminated**

(F) L'analyseur syntaxique a vu une parenthèse gauche à la place de ce qu'il attendait être un terme, donc il recherche la parenthèse droite correspondante, et ne la trouve pas. Il y a des chances que vous ayez oublié des parenthèses nécessaires plus tôt dans la ligne.

**Global symbol "%s" requires explicit package name**

(F) Vous spécifiez « `use strict vars` », ce qui indique que toutes les variables doivent être ou déclarées locales (en utilisant « `my` »), ou explicitement qualifiées pour dire dans quel paquetage la variable globale est déclarée (en utilisant « `::` »).

**goto must have label**

(F) Au contraire de `next` et de `last`, vous n'êtes pas autorisé à aller à une destination non spécifiée. Voir `goto()` dans *perlfunc*.

**Had to create %s unexpectedly**

(S) Une fonction demande un symbole depuis la table des symboles où celle-ci doit déjà exister, mais pour une raison, elle n'existe pas, et a été créée en urgence pour éviter un core dump.

**Hash % %s missing the % in argument %d of %s()**

(D) Seuls les vraiment vieux Perl vous permettaient d'omettre le `%` dans un nom de tableau associatif à certains endroits. C'est maintenant largement obsolète.

**Identifiant too long**

(F) Perl limite la taille des identifiants (noms des variables, fonctions, etc.) à 250 caractères pour les noms simples, et un peu plus pour les noms composés (comme `$A: :B`). Vous avez excédé les limites de Perl. Les versions futures de Perl vont éliminer cette limitation arbitraire.

**Ill-formed logical name |%s| in prime\_env\_iter**

(W) Une alerte spécifique à VMS. Un nom logique a été rencontré lors de la préparation d'itération sur `%ENV`, ce qui viole les règles syntaxiques gouvernant les noms logiques. Comme ils ne peuvent être transmis normalement, ils sont sautés, et n'apparaissent pas dans `%ENV`. Cela peut-être un événement bénin, comme dans certains paquetages de software qui peuvent directement modifier le nom logique des tables et introduire des noms non standards, ou cela peut indiquer que le nom logique a été corrompu.

**Illegal character %s (carriage return)**

(F) Le caractère 'carriage return' a été trouvé en entrée. C'est une erreur, pas une alerte, car le carriage return peut casser une chaîne multi-ligne, incluant le document (e.g., `print <<EOF;`).

Sous Unix, cette erreur est habituellement causée par l'exécution de code Perl – ou par le programme principal, un module, ou un eval sur une chaîne – qui a été transféré à travers le réseau depuis un système non-Unix qui ne convertit pas proprement le format des fichiers textes.

Sous certains systèmes qui utilisent quelque chose d'autre que le `'\n'` pour délimiter les lignes d'un texte, cette erreur peut être causée par la lecture de code Perl depuis un descripteur de fichier qui est en mode binaire (comme positionné par l'opérateur `binmode`).

Dans certains cas, le code Perl en question doit probablement être converti avec quelque chose comme `s/\x0D\x0A?/\n/g` avant de pouvoir être exécuté.

**Illegal division by zero**

(F) Vous essayez de diviser un nombre par 0. Ou quelque chose n'est pas bon dans votre logique, ou vous devez placer une condition pour vous garder de cette entrée dénuée de sens.

**Illegal modulus zero**

(F) Vous essayez de diviser un nombre par 0 pour obtenir le reste. La plupart des nombres ne peuvent faire cela facilement.

**Illegal octal digit**

(F) Vous avez utilisé un 8 ou un 9 dans un nombre octal.

**Illegal octal digit ignored**

(W) Vous avez essayé d'utiliser un 8 ou un 9 dans un nombre octal. L'interprétation du nombre octal est stoppé avant le 8 ou le 9.

**Illegal hex digit ignored**

(W) Vous avez essayé d'utiliser un caractère autre que 0 - 9 ou A - F dans un nombre hexadécimal. L'interprétation du nombre hexadécimal est stoppée avant le caractère illégal.

**Illegal switch in PERL5OPT: %s**

(X) La variable d'environnement PERL5OPT peut être utilisée seulement pour positionner les options suivantes : `-[DIMUdmw]`.

**In string, @%s now must be written as \@%s**

(F) Cela est utilisé pour indiquer sur Perl ??? essaie de deviner si vous voulez un tableau interpolé ou un littéral @. Cela arrive quand la chaîne est utilisée la première fois à l'exécution. Maintenant les chaînes sont parsées au moment de la compilation, et les instances ambiguës de @ doivent être éclaircies, soit en mettant une barre oblique inverse pour indiquer qu'il s'agit d'un littéral, ou en déclarant (ou en utilisant) le tableau dans le programme avant la chaîne (lexicalement). (Parfois cela veut simplement dire qu'un @ peut être interprété comme un tableau.)

**Insecure dependency in %s**

(F) Vous essayez de faire quelque chose que le mécanisme de sécurité sur l'entâchement des variables n'apprécie pas. Ce mécanisme est activé quand vous exécutez un script `setuid` ou `setgid`, ou si vous spécifiez l'option `-T` pour l'activer explicitement. Le mécanisme de sécurité sur l'entâchement des variables marque tous les données dérivées directement ou indirectement de l'utilisateur, qui n'est pas considéré comme allié dans vos transactions. Si une de ces données est utilisée dans une opération « dangereuse », vous obtenez cette erreur. Voir *perlsec* pour plus d'informations.

**Insecure directory in %s**

(F) Vous ne pouvez utiliser `system()`, `exec()`, ou ouvrir un pipe dans un script `setuid` ou `setgid` si `$ENV{PATH}` contient un répertoire qui a les droits en écriture pour tout le monde. Voir *perlsec*.

**Insecure \$ ENV{%s} while running %s**

(F) Vous ne pouvez utiliser `system()`, `exec()`, ou ouvrir un pipe dans un script `setuid` ou `setgid` si `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}` ou `$ENV{BASH_ENV}` sont dérivés de données soumises (ou potentiellement soumises) par l'utilisateur. Le script doit positionner le path à une valeur connue, en utilisant une valeur 'saine'. Voir *perlsec*.

**Integer overflow in hex number**

(S) Le nombre hexadécimal littéral que vous avez spécifié est trop grand pour votre architecture. Sur une architecture 32-bits le plus grand hexadécimal littéral est `0xFFFFFFFF`.

**Integer overflow in octal number**

(S) Le nombre octal littéral que vous avez spécifié est trop grand pour votre architecture. Sur une architecture 32-bits le plus grand octal littéral est `037777777777`.

**Internal inconsistency in tracking vforks**

(S) Une alerte spécifique à VMS. Perl garde une trace du nombre d'appel à `fork` et `exec`, pour déterminer si l'appel courant à `exec` peut affecter le script en cours ou un sous-process (Voir `exec` in *perlvm*). De façon ou d'autre, ce compte a été bousculé, et Perl fait une conjoncture et traite ce `exec` comme une requête pour terminer le script Perl et exécuter la commande spécifiée.

**internal disaster in regexp**

(P) Quelque chose s'est très mal passé dans le parseur d'expressions rationnelles.

**internal error: glob failed**

(P) Il s'est passé quelque chose d'anormal avec le programme externe utilisé avec `glob` et `<*.c>`. Cela veut dire que votre `csh` (C shell) est rompu. Si c'est cela, vous devez changer toutes les variables relatives à `csh` dans `config.sh` : si vous avez `tcsh`, faites pointer les variables vers celui-ci comme si c'était `csh` (ex : `full_csh='/usr/bin/tcsh'`); autrement, mettez-les toutes à blanc. (excepté que `d_csh` devrait être 'undef') pour que Perl pense que `csh` est manquant. Dans ce cas, après avoir édité `config.sh` lancez `./Configure -S` et reconstruisez Perl.

**internal urp in regexp at /%s/**

(P) Quelque chose s'est très mal passé dans le parseur d'expressions rationnelles.

**invalid [] range in regexp**

(F) L'écart spécifié dans la classe de caractères a un caractère minimum supérieur au caractère maximum. Voir *perlre*.

**Invalid conversion in %s: "%s"**

(W) Perl ne comprend pas le format de conversion donné. Voir `sprintf()` dans *perlfunc*.

**Invalid type in pack: '%s'**

(F) Le caractère donné n'est pas un type de « pack » valide. Voir `pack()` dans *perlfunc*. (W) Le caractère donné n'est pas un type de « pack » valide mais est utilisé en étant silencieusement ignoré.

**Invalid type in unpack: '%s'**

(F) Le caractère donné n'est pas un type de « unpack » valide. Voir `unpack()` dans *perlfunc*. (W) Le caractère donné n'est pas un type de « unpack » valide mais est utilisé en étant silencieusement ignoré.

**ioctl is not implemented**

(F) Votre machine apparemment n'implémente pas `ioctl()`, ce qui est vraiment étonnant pour une machine qui supporte le C.

**junk on end of regexp**

(P) Le parseur d'expression rationnelle a été embrouillé.

**Label not found for "last %s"**

(F) Vous nommez une boucle pour en sortir, mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous l'appellez. Voir `last()` dans *perlfunc*.

**Label not found for "next %s"**

(F) Vous nommez une boucle pour continuer, mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous l'appellez. Voir `next()` dans *perlfunc*.

**Label not found for "redo %s"**

(F) Vous nommez une boucle pour recommencer, mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous l'appellez. Voir `redo()` dans *perlfunc*.

**listen() on closed fd**

(W) Vous essayez de faire un `listen()` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de `socket()`? Voir `listen()` dans *perlfunc*.

**Method for operation %s not found in package %s during blessing**

(F) Une tentative a été effectuée pour spécifier une entrée dans une table surchargée qui ne peut résoudre une sous-fonction valide. Voir *overload*.

**Might be a runaway multi-line %s string starting on line %d**

(S) Un avertissement indiquant que l'erreur précédente peut avoir été causée par un délimiteur manquant dans une chaîne ou un motif (pattern), qui se fermerait éventuellement plus tôt dans la ligne courante.

**Misplaced \_ in number**

(W) Un underscore dans une constante décimale n'est pas sur une limite de 3 chiffres.

**Missing \$ on loop variable**

(F) Apparemment vous avez programmé en **cs** trop longtemps. Les variables en Perl sont toujours précédées du \$, au contraire des shells, ou cela peut varier d'une ligne à l'autre.

**Missing comma after first argument to %s function**

(F) Alors que certaines fonctions vous autorisent à spécifier un descripteur de fichier ou un « objet indirect » avant une liste d'argument, celle-ci n'en fait pas partie.

**Missing operator before %s?**

(S) C'est un message donné à titre indicatif, en conjonction avec le message « %s found where operator expected ». Souvent l'opérateur manquant est le point-virgule.

**Missing right bracket**

(F) L'analyseur syntaxique a compté plus d'accolades ouvertes que fermées. En règle générale, vous la trouverez à coté de votre dernière modification.

**Modification of a read-only value attempted**

(F) Vous essayez, directement ou indirectement, de changer la valeur d'une constante. Vous ne pouvez, bien sûr, essayer de faire « 2 = 1 », car le compilateur l'intercepte. Mais il y a une autre façon de faire la même chose :

```
sub mod { $_[0] = 1 }
mod(2);
```

Un autre moyen serait d'affecter à `substr()` ce qui termine la chaîne.

#### **Modification of non-creatable array value attempted, subscript %d**

(F) Vous avez essayé de transformer un élément de tableau en une valeur existante, et l'indice inférieur était probablement négatif, même en comptant à partir de la fin du tableau.

#### **Modification of non-creatable hash value attempted, subscript "%s"**

(P) Vous avez essayé de transformer un élément de tableau de hachage en une valeur existante, et il ne peut être créé pour une raison particulière.

#### **Module name must be constant**

(F) Seul un nom de module est permis comme premier argument d'un "use".

#### **msg%s not implemented**

(F) Vous n'avez pas de messages IPC System V sur votre système.

#### **Multidimensional syntax %s not supported**

(W) Les tableaux multidimensionnels ne s'écrivent pas `$foo[1,2,3]`. Ils s'écrivent `$foo[1][2][3]`, comme en C.

#### **Name "%s::%s" used only once: possible typo**

(W) Erreur typographique souvent vue si une variable n'est initialisée ou utilisée qu'une seule fois. Si vous avez une bonne raison pour faire cela, mentionnez la variable à nouveau pour supprimer ce message. La ligne `use vars` est fournie pour ce contexte.

#### **Negative length**

(F) Vous essayez de faire une opération `read/write/send/recv` avec un buffer de longueur plus petite que 0. C'est difficile à imaginer.

#### **nested \*?+ in regexp**

(F) Vous ne pouvez quantifier un quantifiant sans faire intervenir de parenthèses. Donc les choses comme `**` ou `+*` ou `?*` sont illégales. Remarquez, cependant, que les opérateurs de comparaison `minimum`, `*?`, `+`, et `??` apparaissent comme des quantifiants `nested`, mais ne le sont pas. Voir *perlre*.

#### **No #! line**

(F) L'émulateur `setuid` nécessite que le script possède une ligne formée telle que `#!` même si la machine ne reconnaît pas le constructeur `#!`.

#### **No %s allowed while running setuid**

(F) Certaines opérations sont supposées être dangereuses pour la sécurité pour un script `setuid` ou `setgid` et ne sont pas autorisées à être lancées. Il y a sûrement un autre moyen pour faire ce que vous voulez, si ce moyen n'est pas sécurisé, il est au moins sécurisable. Voir *perlsec*.

#### **No -e allowed in setuid scripts**

(F) Un script `setuid` ne peut être spécifié par l'utilisateur.

#### **No comma allowed after %s**

(F) Une liste d'opérateurs qui contient un descripteur de fichier ou un « object indirect » n'est pas autorisé à avoir une virgule entre lui et les arguments suivants. Autrement, il s'agit juste d'un argument comme un autre.

Une cause possible pour cela est que vous supposez avoir importé une constante dans votre namespace avec `use` ou `import` alors qu'il n'existe pas d'import correspondant, par exemple votre système d'exploitation ne supporte pas une constante particulière. En espérant que vous avez bien utilisé une liste explicite des constantes importées que vous vous attendez à voir, voir `use()` dans *perlfunc* et `import()` dans *perlfunc*. Comme une liste explicite d'import aurait probablement intercepté cette erreur plus tôt, cela ne remédie pas au fait que votre système d'exploitation ne supporte pas cette constante. Peut-être avez-vous une erreur de frappe dans vos constantes des listes de symboles d'import de `use` ou `import` ou dans le nom de constante à la ligne où l'erreur est apparue ?

#### **No command into which to pipe on command line**

(F) Une erreur spécifique à VMS. Perl handles its own command line??? redirection, and found a '|' at the end of the command line, so it doesn't know where you want to pipe the output from this command.

#### **No DB::DB routine defined**

(F) Le code qui est en train d'être exécuté avec l'option `-d`, mais pour une raison quelconque le fichier `perl5db.pl` (ou un similaire) ne définit pas une routine à appeler en début de chaque déclaration. Ce qui est étrange, car le fichier a été chargé automatiquement, et ne peut passer le require s'il ne l'a pas parsé correctement.

**No dbm on this machine**

(P) C'est compté comme une erreur interne, car chaque machine doit fournir dbm de nos jours, car Perl est fourni avec SDBM. Voir SDBM\_File.

**No DBsub routine**

(F) Le code est exécuté avec l'option **-d**, mais pour une raison quelconque le fichier perl5db.pl (ou un similaire) ne définit pas une routine DB::sub à appeler en début de chaque appel de sous-routines ordinaires.

**No error file after 2> or 2>> on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirection de ligne de commandes, et a trouvé sur l'entrée standard '2>' ou '2>>', mais ne trouve pas le nom du fichier dans lequel écrire les données destinée à stderr.

**No input file after < on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commandes, et a trouvé sur l'entrée standard '<', mais ne trouve pas le nom du fichier dans lequel écrire les données destinées à stdin.

**No output file after > on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commandes, et a trouvé sur l'entrée standard '>' en fin de ligne, donc il ne trouve pas où vous voulez rediriger votre sortie.

**No output file after > or >> on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commandes, et a trouvé sur l'entrée standard a '>' ou '>>', mais ne trouve pas le nom du fichier dans lequel écrire les données destinées à stdout.

**No Perl script found in input**

(F) Vous appelez `perl -x`, mais aucune ligne n'est trouvée dans le fichier avec `#!` et contenant le mot « perl ».

**No setregid available**

(F) Configure n'a rien trouvé qui ressemble à l'appel de `setregid()` pour votre système.

**No setreuid available**

(F) Configure n'a rien trouvé qui ressemble à l'appel de `setreuid()` pour votre système.

**No space allowed after -I**

(F) L'argument de **-I** doit suivre **-I** immédiatement après sans qu'aucun espace n'intervienne.

**No such array field**

(F) Vous essayez d'accéder à un tableau comme avec un tableau associatif, mais le nom de champ utilisé n'est pas défini. Le hash à l'index 0 doit contenir tous les noms de champs valides pour un tableau d'indices pour que cela fonctionne.

**No such field "%s" in variable %s of type %s**

(F) Vous essayez d'accéder à un champ d'une variable typée où le type ignore le nom du champ. Le nom du champ est recherché dans le tableau associatif `%FIELDS` dans le paquetage `type` à la compilation. Le tableau `%FIELDS` est généralement utilisé avec le pragma `'fields'`.

**No such pipe open**

(P) Une erreur spécifique à VMS. La routine interne `my_pclose()` a essayé de fermer un pipe qui n'a pas été ouvert. Ceci peut être capturé plus tôt comme une tentative de fermer un descripteur non ouvert.

**No such signal: SIG%s**

(W) Vous spécifiez un nom de signal comme une sous-fonction d'un sous-script à `%SIG` qu'il ne reconnaît pas. Faire `kill -l` dans votre shell pour voir les noms de signaux valides sur votre système.

**Not a CODE reference**

(F) Perl essaye d'évaluer une référence vers une valeur de code (c'est cela, une sous-fonction), mais trouve une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref()` pour trouver de quelle référence il s'agit exactement. Voir *perlref*.

**Not a format reference**

(F) Je ne suis pas sûr de la manière dont vous dirigez la génération d'une référence vers un format anonyme, mais cela indique que vous l'avez fait, et cela n'existe pas.

**Not a GLOB reference**

(F) Perl essaye d'évaluer une référence vers un « typeglob » (c'est cela, un symbole d'entrée de table qui ressemble à `*foo`), mais trouve une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref()` pour trouver de quelle référence il s'agit exactement. Voir *perlref*.

**Not a HASH reference**

(F) Perl essaye d'évaluer une référence vers une valeur d'un tableau associatif mais trouve une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref()` pour trouver de quelle référence il s'agit exactement. Voir *perlref*.



**Not a perl script**

(F) L'émulateur setuid requiert que le script ait une ligne `#!` même sur les machines qui ne supportent pas le constructeur `#!`. La ligne doit au moins mentionner `perl`.

**Not a SCALAR reference**

(F) Perl essaye d'évaluer une référence vers un scalaire, mais trouve une référence vers quelque chose d'autre. Vous pouvez utiliser la fonction `ref()` pour trouver de quelle référence il s'agit exactement. Voir *perlref*.

**Not a subroutine reference**

(F) Perl essaye d'évaluer une référence vers une valeur de code (c'est cela, une sous-fonction), mais trouve une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref()` pour trouver de quelle référence il s'agit exactement. Voir *perlref*.

**Not a subroutine reference in overload table**

(F) Une tentative a été faite pour spécifier une entrée dans une table surchargée qui ne pointe pas vers une sous-fonction valide. Voir *overload*.

**Not an ARRAY reference**

(F) Perl essaye d'évaluer une référence vers une valeur de tableau, mais trouve une référence vers quelque chose de différent. Vous pouvez utiliser la fonction `ref()` pour trouver de quel type de référence il s'agit vraiment. Voir *perlref*.

**Not enough arguments for %s**

(F) La fonction nécessite plus d'arguments que vous n'avez spécifiés.

**Not enough format arguments**

(W) Le format spécifie plus de champs d'images que la ligne suivante ne lui en fournit. Voir *perlform*.

**Null filename used**

(F) Vous ne pouvez charger un fichier avec un nom nul, car sur certaines machines cela veut dire le répertoire courant ! Voir `require()` dans *perlfunc*.

**Null picture in formline**

(F) Le premier argument de `formline` doit être un format de spécification d'image valide. Cet argument est vide, ce qui veut probablement dire que vous lui avez soumis une valeur non initialisée. Voir *perlform*.

**NULL OP IN RUN**

(P) Une fonction interne a appelé `run()` avec un pointer null opcode.

**Null realloc**

(P) Une tentative a eu lieu pour réallouer NULL.

**NULL regexp argument**

(P) La fonction interne de recherche de motif a été soufflée.

**NULL regexp parameter**

(P) La fonction interne de recherche de motif a été soufflée. (Ndt : Hors de sa course dans la version originale)

**Number too long**

(F) Perl limite la représentation des nombres décimaux dans le programme à 250 caractères. Vous avez dépassé la limite. Les versions futures de Perl vont éliminer cette limitation arbitraire. En attendant, essayez d'utiliser la notation scientifique (ex. « `1e6` » au lieu de « `1_000_000` »).

**Odd number of elements in hash assignment**

(S) Vous avez spécifié un nombre impair d'éléments pour initialiser votre tableau indexé, ce qui est bizarre car un tableau indexé est utilisé avec des paires clé/valeur.

**Offset outside string**

(F) Vous essayez de faire une opération `read/write/send/recv` operation avec un offset qui pointe en dehors du buffer. Cela est difficile à imaginer. La seule exception à cela est que `sysread()` past the buffer will extend the buffer and zero pad the new area.???

**oops: oopsAV**

(S) Un avertissement interne que la grammaire est vissée vers le haut.???

**oops: oopsHV**

(S) Un avertissement interne que la grammaire est vissée vers le haut.

**Operation '%s': no method found, %s**

(F) une tentative a été faite pour surcharge d'opérations pour laquelle il n'existe pas de descripteur de fichier défini. Alors que certains descripteurs peuvent être automatiquement régénérés dans les conditions d'autres descripteurs, il n'y a pas de descripteur par défaut pour quelque opération que ce soit, à moins que la clé `fallback` ne soit spécifiée à vrai. Voir *overload*.

**Operator or semicolon missing before %s**

(S) Vous utilisez une variable ou un appel de fonction là où le parseur s'attend à trouver un opérateur. Le parseur suppose que vous pensiez vraiment utiliser un opérateur, mais cela est fortement vu comme incorrect. Par exemple, si vous dites « \*foo \*foo » cela est interprété comme si vous disiez « \*foo \* 'foo' ».

**Out of memory for yacc stack**

(F) Le parseur yacc recherche à agrandir sa pile donc il peut continuer, mais realloc() ne veut pas lui donner plus de mémoire, virtuelle ou autre.

**Out of memory during request for %s**

(X|F)(F) La fonction malloc() a retourné 0, ce qui indique un manque de mémoire (ou de mémoire virtuelle) pour satisfaire la requête. La requête est jugée petite, ce qui rend la possibilité de capturer l'erreur dépendante de la façon dont Perl a été compilé. Par défaut, cela ne peut être capturé. Cependant, si compilé à cet effet, Perl peut utiliser le contenu de \$^M comme une sortie d'urgence après le die() et son message. Dans ce cas l'erreur peut-être capturée.

**Out of memory during "large" request for %s**

(F) La fonction malloc() a retourné 0, ce qui indique un manque de mémoire (ou de mémoire virtuelle) pour satisfaire la requête. Cependant, la requête a été jugée assez large (par défaut 64K), donc la possibilité de capturer cette erreur est permise.

**Out of memory during ridiculously large request**

(F) Vous ne pouvez allouer plus de  $2^{31}+1$  octets. Cette erreur est généralement causée par une faute de frappe dans le programme Perl. Ex, \$arr[time] au lieu de \$arr[\$time].

**page overflow**

(W) Un appel simple de write() a produit plus de lignes que peut contenir une page. Voir *perform*.

**panic: ck\_grep**

(P) Echec d'un test de cohérence lors de la compilation d'un grep.

**panic: ck\_split**

(P) Echec d'un test de cohérence lors de la compilation d'un split.

**panic: corrupt saved stack index**

(P) La sauvegarde de la pile a été appelée pour restaurer plus de valeurs qu'il n'y en a dans la pile.

**panic: die %s**

(P) On est passé du contexte de pile à un contexte de eval, pour finalement découvrir qu'on n'est pas dans un contexte d'eval.

**panic: do\_match**

(P) La fonction interne pp\_match() a été appelé avec des données opérationnelles non valides.

**panic: do\_split**

(P) Quelque chose de grave est arrivé lors de la préparation du split.

**panic: do\_subst**

(P) La fonction interne pp\_subst() a été appelé avec des données opérationnelles non valides.

**panic: do\_trans**

(P) La fonction interne do\_trans() a été appelé avec des données opérationnelles non valides.

**panic: frexp**

(P) La fonction frexp() de la library a échoué, rendant le printf("%f") impossible.

**panic: goto**

(P) On a déchargé le contexte de pile vers le contexte avec l'étiquette (label) spécifiée, et ensuite découvert que c'était une étiquette que l'on ne sait pas joindre.

**panic: INTERCASEMOD**

(P) L'analyseur syntaxique est arrivé dans ce mauvais état au moment d'un case modifier.???

**panic: INTERPCONCAT**

(P) L'analyseur syntaxique est dans un mauvais état après avoir parsé une chaîne de caractères avec des parenthèses.

**panic: last**

(P) On a déchargé le contexte de pile vers un contexte de bloc, et découvert par la suite que ce n'est pas un contexte de bloc.

**panic: leave\_scope clearsv**

(P) Une variable lexicale est maintenant en lecture seule d'une manière ou d'une autre dans le bloc.

**panic: leave\_scope inconsistency**

(P) La sauvegarde de pile n'est sûrement plus synchronisée. À la fin, il y a un type enum invalide en haut du tout.

**panic: malloc**

(P) Quelque chose a demandé un nombre négatif d'octets dans malloc.

**panic: mapstart**

(P) Le compilateur s'est embrouillé avec la fonction map().

**panic: null array**

(P) L'un des tableaux internes de fonctions a passé un pointeur AV null

**panic: pad\_alloc**

(P) Le compilateur s'est embrouillé au sujet de quelle zone de travail où affecter les valeurs temporaires et lexicales.

**panic: pad\_free curpad**

(P) Le compilateur s'est embrouillé au sujet de quelle zone de travail où affecter les valeurs temporaires et lexicales.

**panic: pad\_free po**

(P) Un décalage incorrect de zone de travail a été détecté en interne.

**panic: pad\_reset curpad**

(P) Le compilateur s'est embrouillé au sujet de quelle zone de travail où affecter les valeurs temporaires et lexicales.

**panic: pad\_sv po**

(P) Un décalage incorrect de zone de travail a été détecté en interne.

**panic: pad\_swipe curpad**

(P) Le compilateur s'est embrouillé au sujet de quelle zone de travail où affecter les valeurs temporaires et lexicales.

**panic: pad\_swipe po**

(P) Un décalage incorrect de zone de travail a été détecté en interne.

**panic: pp\_iter**

(P) L'itérateur foreach n'a pas été appelé dans un contexte de boucle.

**panic: realloc**

(P) Quelque chose a demandé un nombre négatif d'octets à realloc.

**panic: restartop**

(P) Une routine interne a demandé un goto (ou quelque chose de semblable), et n'a pas fourni de destination.

**panic: return**

(P) On est passé du contexte de pile à un contexte de sous-fonction ou eval, pour finalement découvrir qu'on n'est pas dans un contexte d'eval ou de sous-fonction.

**panic: scan\_num**

(P) scan\_num() a été appelé avec quelque chose qui n'est pas un nombre.

**panic: sv\_insert**

(P) La routine sv\_insert() a été utilisé pour enlever plus de chaînes qu'il n'en existe.

**panic: top\_env**

(P) Le compilateur s'attendait à faire un goto, ou quelque chose d'équivalent.

**panic: yylex**

(P) L'analyseur syntaxique est dans un état invalide pendant qu'il était en train de procéder à un case.

**Parentheses missing around "%s" list**

(W) Vous dites quelque chose comme

```
my $foo, $bar = @_;
```

alors que vous pensez

```
my ($foo, $bar) = @_;
```

Souvenez-vous que « my » et « local » doivent être délimités par des parenthèses.

**Perl %3.3f required--this is only version %s, stopped**

(F) Le module en question utilise une possibilité offerte par une version plus récente que celle exécutée actuellement. Depuis combien de temps votre Perl n'a-t-il pas été mis à jour ? Voir require() dans perlfunc.

**Permission denied**

(F) L'émulateur suidperl voit qui vous êtes et interdit l'exécution.

**pid %d not a child**

(W) Une alerte spécifique à VMS. Waitpid() a été appelée pour attendre un processus qui n'est pas un sous-process du process courant. Même si cela est bien sous la perspective VMS, cela n'est probablement pas ce que vous voulez.

**POSIX getpgrp can't take an argument**

(F) Votre compilateur C utilise la fonction POSIX getpgrp(), qui ne prend pas d'argument, au contraire de la version BSD, qui prend un pid.

**Possible attempt to put comments in qw() list**

(W) Une liste par qw() ne contient que des éléments séparés par des espaces; comme avec les chaînes littérales, les caractères de commentaires ne sont pas ignorés, mais au lieu de ça traités comme des données littérales. (Vous pouvez utiliser un délimiteur différent que les parenthèses indiquées ici. Les crochets sont fréquemment utilisés.)

Vous avez probablement écrit quelque chose comme cela :

```
@list = qw(
 a # a comment
 b # another comment
);
```

alors qu'il devrait être écrit comme cela :

```
@list = qw(
 a
 b
);
```

Si vous voulez vraiment des commentaires, construisez votre liste à l'ancienne façon, avec des cotes et des virgules :

```
@list = (
 'a', # un commentaire
 'b', # un autre commentaire
);
```

**Possible attempt to separate words with commas**

(W) Les listes qw() contiennent des éléments séparés par des espaces. Donc, les virgules ne sont pas nécessaires pour séparer les éléments. (Vous pouvez utiliser un délimiteur différent des parenthèses montrées ici ; les crochets sont fréquemment utilisés.)

Vous avez écrit probablement quelque chose comme ça :

```
qw! a, b, c !;
```

Ce qui place les caractères virgules comme éléments de la liste. Écrivez ceci sans virgule si vous ne voulez pas que cela apparaisse dans vos données :

```
qw! a b c !;
```

**Possible memory corruption: %s overflowed 3rd argument**

(F) Un appel à ioctl() ou fcntl() a retourné plus que ce Perl peut traiter. Perl consacre un tampon raisonnable, mais place un octet sentinelle à la fin du tampon au cas où. Le bit sentinelle a été dépassé, et Perl suppose que sa mémoire est maintenant corrompue. Voir ioctl() dans *perlfunc*.

**Precedence problem: open %s should be open(%s)**

(S) L'ancienne construction irrégulière

```
open FOO || die;
```

est maintenant mal interprétée comme

```
open(FOO || die);
```

à cause de la stricte régularisation de la grammaire de Perl 5 en opérateurs unaires et listes. (L'ancien était un peu des deux.) Vous devez mettre une parenthèse autour du descripteur de fichier, ou utiliser le nouvel opérateur « or » à la place de « || ».

**print on closed filehandle %s**

(W) Le descripteur de fichier dans lequel vous essayez d'imprimer a été fermé quelque temps auparavant. Vérifiez votre flux de données.

**printf on closed filehandle %s**

(W) Le descripteur de fichier dans lequel vous essayez d'écrire a été fermé quelque temps auparavant. Vérifiez votre flux de données.

**Probable precedence problem on %s**

(W) Le compilateur a trouvé un simple mot là où il attendait un opérateur conditionnel, ce qui indique souvent que le `||` ou `&&` a été parsé comme une partie du dernier argument du constructeur, par exemple :

```
open F00 || die;
```

**Prototype mismatch: %s vs %s**

(S) La sous-fonction qui est en train d'être déclarée ou définie a été précédemment déclarée ou définie avec un prototype différent.

**Range iterator outside integer range**

(F) Un (ou deux) arguments numériques de l'opérateur d'échelles `<..>` est en dehors des bornes qui peuvent être définies par des entiers en interne. Un moyen de détourner cela est de forcer Perl à utiliser les chaînes de caractères « magiques » incrémentées en ajoutant `<0>` à vos nombres.

**Read on closed filehandle <%s>**

(W) Le descripteur de fichier a été fermé avant cette opération. Vérifiez votre flux logique de données.

**Reallocation too large: %lx**

(F) Vous ne pouvez allouer plus de 64K sur une machine MS-DOS.

**Recompile perl with -DDEBUGGING to use -D switch**

(F) Vous ne pouvez utiliser l'option `-D` à moins que le code pour produire la sortie désirée ait été compilé dans Perl, ce qui implique certaines contraintes non disponibles dans votre version de Perl.

**Recursive inheritance detected in package '%s'**

(F) Au moins 100 niveaux d'héritage ont été utilisés. Cela indique probablement une boucle inattendue dans votre hiérarchie de classes.

**Recursive inheritance detected while looking for method '%s' in package '%s'**

(F) Au moins 100 niveaux d'héritage ont été rencontrés lors de l'appel de méthodes. Cela indique probablement une boucle inattendue dans votre hiérarchie de classes.

**Reference found where even-sized list expected**

(W) Vous avez donné une référence unique là où Perl s'attend à avoir une liste avec un même nombre d'éléments (pour affectation à un tableau associatif). Cela veut dire que vous utilisez le constructeur `anon hash` alors que vous vous attendiez à utiliser `parens`. Dans tous les cas, un tableau associatif (hash) nécessite une **paire** clef/valeur.

```
%hash = { one => 1, two => 2, }; # MAUVAIS
%hash = [qw/ an anon array /]; # MAUVAIS
%hash = (one => 1, two => 2,); # correct
%hash = qw(one 1 two 2); # correct
```

**Reference miscount in sv\_replace()**

(W) La fonction interne `sv_replace()` a levé une nouvelle SV avec un nombre de références différent de 1.

**regexp \*+ operand could be empty**

(F) La partie d'un motif (regexp) sujet à l'opérateur de quantification `*` ou `+` doit être une chaîne vide.

**regexp memory corruption**

(P) Le moteur d'expression rationnelle a été mis en confusion par ce que l'expression rationnelle lui a donnée.

**regexp out of space**

(P) Une erreur « qui ne peut pas arriver », car `safemalloc()` aurait dû la capturer plus tôt.

**regexp too big**

(F) L'implémentation courante des expressions rationnelles utilise des shorts comme offset d'adresse dans une chaîne. Malheureusement cela veut dire que si l'expression rationnelle compilée est plus longue que 32767, il s'arrête. Généralement quand vous voulez une expression rationnelle aussi longue, il est mieux de passer par des déclarations multiples. Voir *perlre*.

**Reversed %s= operator**

(W) Vous écrivez votre opérateur d'affectation à l'envers. Le `=` doit toujours arriver en dernier, pour éviter l'ambiguïté avec les opérateurs unaires suivants.

**Runaway format**

(F) Votre format contient la séquence `~~` repeter-jusqu'à-la-sequence-de-blanc, mais cela produit 200 lignes au moins, et la 200<sup>e</sup> ligne apparaît exactement comme la 199<sup>e</sup>. Apparemment vous n'avez pas arrangé les arguments pour qu'ils s'épuisent, soit en utilisant `^` au lieu de `@` (pour les variables scalaires), soit en faisant un `shift` ou un `pop` (pour les tableaux). Voir *perlfm*.

**Scalar value @ %s[%s] better written as \$ %s[%s]**

(W) Vous avez utilisé une tranche de tableau (indiqué par `@`) pour sélectionner un élément unique d'un tableau. Généralement, il est mieux de demander pour une valeur scalaire (indiquée par `$`). La différence est que `$foo[&bar]` se comporte toujours comme un scalaire, à la fois au moment de l'affectation et quand vous l'évaluez, alors que `@foo[&bar]` se comporte comme une liste lorsque vous l'affectez, et fournit une liste dans un sous-script, ce qui peut faire des choses étranges si vous vous attendiez à seulement un sous-script. En d'autres termes, si vous espérez actuellement traiter l'élément de tableau associatif comme une liste, vous devez regarder dedans comment les références fonctionnent, car Perl ne va pas faire la conversion entre les scalaires et les listes pour vous. Voir *perlref*.

**Scalar value @ %s{ %s} better written as \$ %s{ %s}**

(W) Vous avez utilisé une tranche de tableau (indiqué par `@`) pour sélectionner un élément unique d'un tableau. Généralement, il est mieux de demander pour une valeur scalaire (indiquée par `$`). La différence est que `$foo{&bar}` se comporte toujours comme un scalaire, à la fois au moment de l'affectation et quand vous l'évaluez, alors que `@foo{&bar}` se comporte comme une liste lorsque vous l'affectez, et fournit une liste dans un sous-script, ce qui peut faire des choses étranges si vous vous attendiez à seulement un sous-script.

En d'autres termes, si vous espérez actuellement traiter l'élément de tableau associatif comme une liste, vous devez regarder dedans comment les références fonctionnent, car Perl ne va pas faire la conversion entre les scalaires et les listes pour vous. Voir *perlref*.

**Script is not setuid/setgid in suidperl**

(F) Bizarrement, le programme `suidperl` a été invoqué dans un script qui n'a pas de bit `setuid` ou `setgid` bit positionné. Cela n'a vraiment pas de sens.

**Search pattern not terminated**

(F) L'analyseur syntaxique ne peut trouver le délimiteur final du constructeur de `//` ou `m{ }`. Souvenez-vous que les parenthèses comptent les niveaux voisins. Manquer un `$` d'une variable `$m` peut causer cette erreur.

**%sseek() on unopened file**

(W) Vous essayez d'utiliser un `seek()` ou `sysseek()` sur un descripteur de fichier qui n'a jamais été ouvert ou qui a été fermé depuis.

**select not implemented**

(F) Cette machine n'implémente pas l'appel système `select()`.

**sem %s not implemented**

(F) Vous n'avez pas de sémaphore IPC System V sur votre système.

**semi-panic: attempt to dup freed string**

(S) La routine interne `newSVsv()` a été appelée pour dupliquer un scalaire qui a été marqué comme libre depuis.

**Semicolon seems to be missing**

(W) Une erreur de syntaxe a été probablement causée par un point-virgule manquant ou un autre opérateur manquant, comme une parenthèse.

**Send on closed socket**

(W) Le descripteur de fichier a été fermé avant cette opération. Vérifiez votre flux logique de données.

**Sequence (? incomplete**

(F) Une expression rationnelle est terminée par une extension incomplète. (?). Voir *perlre*.

**Sequence (?#... not terminated**

(F) Une expression rationnelle doit se terminer par une parenthèse fermée. Les parenthèses incluses ne sont pas autorisées. Voir *perlre*.

**Sequence (?%s...) not implemented**

(F) L'extension de l'expression rationnelle proposée a le mot réservé mais n'a pas encore été écrite. Voir *perlre*.

**Sequence (?%s...) not recognized**

(F) Vous utilisez une expression rationnelle qui n'a pas de sens. Voir *perlre*.

**Server error (500 Server error)**

Connu sous le nom de « 500 Server error ». **Ceci est une erreur CGI, pas une erreur Perl.** Vous devez être sûr que votre script est exécutable, accessible par l'utilisateur CGI qui lance le script (qui n'est probablement pas

l'utilisateur que vous utilisez pour faire vos tests), que vous n'utilisez pas de variables d'environnement (comme PATH) que l'utilisateur CGI ne va pas avoir, et que vous n'utilisez pas de ressources injoignables pour le serveur. Voir pour plus d'information :

```
http://www.perl.com/perl/faq/idiots-guide.html
http://www.perl.com/perl/faq/perl-cgi-faq.html
ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq
http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html
```

#### **setegid() not implemented**

(F) Vous essayez d'affecter \$), et votre système d'exploitation ne supporte pas l'appel système setegid() (ou un équivalent), ou du moins c'est ce que Configure pense.

#### **seteuid() not implemented**

(F) Vous essayez de référencer \$>, et votre système d'exploitation ne supporte pas l'appel système seteuid() (ou équivalent), ou au moins Configure n'y a pas pensé.

#### **setrgid() not implemented**

(F) Vous essayez de référencer \$(, et votre système d'exploitation ne supporte pas l'appel système setrgid() (ou équivalent), ou au moins Configure n'y a pas pensé.

#### **setruid() not implemented**

(F) Vous essayez de référencer \$<, et votre système d'exploitation ne supporte pas l'appel système setruid() (ou équivalent), ou au moins Configure n'y a pas pensé.

#### **Setuid/gid script is writable by world**

(F) L'émulateur setuid ne va pas lancer un script qui est modifiable par le monde, car le monde peut très bien déjà l'avoir modifié.

#### **shm%s not implemented**

(F) Vous n'avez pas de System V shared memory IPC sur votre système.

#### **shutdown() on closed fd**

(W) Vous essayez de faire un shutdown sur socket fermée. Cela paraît un peu superflu.

#### **SIG%s handler "%s" not defined**

(W) Le gestionnaire de signal nommé dans %SIG n'existe pas. Peut-être l'avez-vous défini dans le mauvais paquetage ?

#### **sort is now a reserved word**

(F) Un message d'erreur ancien que personne ne lancera plus. Car avant que 'sort' soit un mot-clef, on l'utilisait parfois comme un descripteur de fichier.

#### **Sort subroutine didn't return a numeric value**

(F) Une comparaison avec la fonction sort() doit retourner un nombre. Cela est arrivé en utilisant <=> ou cmp, ou en ne les utilisant pas correctement. Voir sort() dans *perlfunc*.

#### **Sort subroutine didn't return single value**

(F) Une comparaison faite avec le sous-programme sort() peut ne pas retourner une liste avec plus ou moins d'un élément. Voir sort() dans *perlfunc*.

#### **Split loop**

(P) Un split boucle indéfiniment. (Évidemment, un split ne peut boucler plus de fois qu'il y a de caractères en entrée, ce qui est arrivé.) Voir split() dans *perlfunc*.

#### **Stat on unopened file <%s>**

(W) Vous essayez d'utiliser la fonction stat() (ou une fonction de test de fichier équivalent) sur un descripteur de fichier qui n'a jamais été ouvert ou qui a été fermé depuis.

#### **Statement unlikely to be reached**

(W) Vous faites un exec() avec certaines déclarations après autre qu'un die()???. Cela est toujours une erreur car exec() ne retourne jamais rien à moins d'une erreur. Vous voulez probablement utiliser system() à la place, qui lui retourne. Pour supprimer ce warning, mettez le exec dans un bloc tout seul.

#### **Stub found while resolving method '%s' overloading '%s' in package '%s'**

(P) Surcharger la résolution au dessus de l'arbre @ISA peut être rompue par l'importation de stubs. Les stubs ne peuvent jamais être implicitement créés, mais un appel explicite à can peut rompre cela.

#### **Subroutine %s redefined**

(W) Vous redéfinissez un sous-programme. Pour supprimer ce message, faites

```
{
 local $^W = 0;
 eval "sub name { ... }";
}
```

**Substitution loop**

(P) La substitution boucle indéfiniment. (Évidemment, une substitution ne peut itérer plus de fois qu'il y a de caractères en entrée, ce qui est arrivé.) Voir la discussion sur les substitutions dans Opérateurs apostrophe et type apostrophe in *perlop*.

**Substitution pattern not terminated**

(F) L'analyseur syntaxique ne peut trouver le délimiteur intérieur du constructeur de `s///` ou `s{ }{ }`. Souvenez-vous que les parenthèses délimitent le nombre de voisins. Un `$` manquant sur une variable `$s` peut causer cette erreur.

**Substitution replacement not terminated**

(F) L'analyseur syntaxique ne peut trouver le délimiteur intérieur du constructeur de `s///` ou `s{ }{ }`. Souvenez-vous que les parenthèses délimitent le nombre de voisins. Un `$` manquant sur une variable `$s` peut causer cette erreur.

**substr outside of string**

(S),(W) Vous essayez de faire référence à la fonction `substr()` qui pointe en dehors d'une chaîne. En fait la valeur absolue de l'offset est plus grand que la longueur de la chaîne. Voir `substr()` dans *perlfunc*. Cette alerte est impérative si `substr` est utilisé dans un contexte de lvalue (comme opérateur du côté gauche de l'affectation, ou comme argument à une sous-fonction par exemple.)

**suidperl is no longer needed since %s**

(F) Votre Perl a été compilé avec `-DSETUID_SCRIPTS_ARE_SECURE_NOW`, mais une version de l'émulateur `setuid` est tout de même arrivée à se lancer.

**syntax error**

(F) Cela veut probablement dire que vous avez une erreur de syntaxe. Les raisons les plus probables sont :

```
Un mot clef est mal écrit.
Un point-virgule est manquant.
Une virgule est manquante.
Une parenthèse ouverte ou fermée est manquante.
Un crochet ouvrant ou fermant est manquant.
Il manque une cote.
```

Souvent, il y a un autre message associé avec l'erreur de syntaxe qui donne plus d'information. (Des fois cela aide d'activer `-w`.) Le message d'erreur en lui-même dit souvent à quelle ligne il s'est arrêté. Des fois l'erreur actuelle est bien avant, car Perl est fort pour la compression d'entrée au hasard. Occasionnellement le numéro de ligne peut être inexact, et la seule manière de savoir ce qui ce passe est d'appeler de façon répétitive `perl -c`, en découpant à la moitié du programme à chaque fois pour voir où l'erreur apparaît. Une sorte de version cybernétique de 20 questions.

**syntax error at line %d: '%s' unexpected**

(A) Vous lancez accidentellement votre script à travers le Bourne shell à la place de Perl. Vérifiez la ligne `#!`, ou positionnez manuellement votre script dans Perl vous-même.

**System V %s is not implemented on this machine**

(F) Vous essayez de faire quelque chose avec une fonction commençant par « `sem` », « `shm` », ou « `msg` » mais ce System V IPC n'est pas implémenté sur votre machine. Sur certaines machines la fonctionnalité peut exister mais n'est pas configurée. Consulter votre support système.

**Syswrite on closed filehandle**

(W) Le descripteur de fichier dans lequel vous écrivez a été fermé quelques????? auparavant. Vérifiez votre flux logique de données.

**Target of goto is too deeply nested**

(F) Vous essayez d'atteindre une étiquette en utilisant `goto`, une étiquette qui est trop loin pour que Perl puisse l'atteindre. Perl vous fait une faveur en vous le refusant.

**tell() on unopened file**

(W) Vous essayez d'utiliser la fonction `tell()` sur un descripteur de fichier qui n'a jamais été ouvert ou qui a été fermé depuis.

**Test on unopened file <%s>**

(W) Vous essayez d'invoquer un opérateur de test de fichier sur un descripteur de fichier qui n'est pas ouvert. Vérifiez votre logique. Voir aussi `-X` in *perlfunc*.



**That use of \$ [ is unsupported**

(F) L'affectation de \$[ est maintenant strictement réglementée, et interprétée comme une directive du compilateur. Vous devez maintenant seulement avoir une solution parmi celles-ci :

```
$[= 0;
$[= 1;
...
local $[= 0;
local $[= 1;
...
```

Cela est pour prévenir le problème d'un module changeant la base du tableau depuis un autre module par inadvertance. Voir \$[ in *perlvar*.

**The %s function is unimplemented**

La fonction indiquée n'est pas implémentée sur cette architecture, en accord avec les choix de Configure.

**The crypt() function is unimplemented due to excessive paranoia**

(F) Configure ne peut trouver la fonction crypt() sur votre machine, probablement parce que votre vendeur ne l'a pas fournie, probablement pas qu'il pense que c'est un secret, ou du moins ils prétendent que ça continue d'être le cas. Et si vous citez mes paroles, je les dénierai.

**The stat preceding -l \_ wasn't an lstat**

(F) Cela n'a pas de sens de tester le tampon courant de stat pour un lien symbolique si le dernier stat qui a écrit dans le tampon a déjà passé le lien symbolique pour obtenir le fichier réel. Utilisez un autre nom de fichier à la place.

**times not implemented**

(F) Votre version de librairie C ne fait pas apparemment de times(). Je suspecte que vous n'êtes pas sous Unix.

**Too few args to syscall**

(F) Il doit y avoir au moins un argument à la fonction syscall() pour spécifier l'appel système à appeler, étourdi.

**Too late for "-T" option**

(X) La ligne #! (ou l'équivalent local) dans un script Perl contient l'option **-T**, mais Perl n'a pas été invoquée avec **-T** en ligne de commande. C'est une erreur car, quand Perl découvre le **-T** dans un script, il est trop tard pour tout teinter???? dans l'environnement. Donc Perl rend la main.

Si le script Perl a été exécuté comme une commande utilisant le #! mécanisme (ou son équivalent local), cette erreur peut être éventuellement corrigée en éditant la ligne #! pour que l'option **-T** soit une part du premier argument de Perl : ex. Changer `perl -n -T en perl -T -n`.

Si le script Perl a été exécuté avec `perl scriptname`, alors l'option **-T** doit apparaître sur la ligne de commande : `perl -T scriptname`.

**Too late for "-%s" option**

(X) La ligne #! (ou l'équivalent local) dans un script Perl contient l'option **-M** ou **-m**. C'est une erreur car les options **-M** et **-m** ne sont pas prévues pour être utilisées dans un script. Utilisez `use` à la place.

**Too many ('s****Too many )'s**

(A) Vous lancez accidentellement votre script à travers **cs** au lieu de Perl. Vérifiez la ligne #!, ou soumettez manuellement votre script à Perl vous-même.

**Too many args to syscall**

(F) Perl supporte un maximum de seulement 14 arguments pour syscall().

**Too many arguments for %s**

(F) La fonction demande moins d'arguments que ceux que vous avez spécifiés.

**trailing \ in regexp**

(F) L'expression rationnelle se termine par une bloque oblique verticale toute seule (au lieu de deux). Voir *perlre*.

**Transliteration pattern not terminated**

(F) L'analyseur syntaxique ne peut trouver le délimiteur intérieur d'une construction de type `tr///` ou `tr[][]` ou `y///` ou `y[][]`. Le \$ manquant devant les variables \$`tr` ou \$`y` peuvent causer cette erreur.

**Transliteration replacement not terminated**

(F) L'analyseur syntaxique ne peut trouver le délimiteur final d'une construction de type `tr///` ou `tr[][]`.

**truncate not implemented**

(F) Votre machine n'implémente pas de mécanisme de troncation de fichier que Configure peut reconnaître.

**Type of arg %d to %s must be %s (not %s)**

(F) Cette fonction requiert que l'argument de cette position soit d'un certain type. Les tableaux doivent être @NAME ou @{EXPR}. Les tableaux associatifs doivent être %NAME ou %{EXPR}. Les effacements de références de façon implicite ne sont pas permis. Utilisez la forme {EXPR} comme effacements de références explicites. Voir *perlref*.

**umask: argument is missing initial 0**

(W) Un umask de 222 est incorrect. Cela peut-être 0222, car les littéraires octaux commencent toujours par 0 en Perl, comme en C.

**umask not implemented**

(F) Votre machine ne semble pas implémenter la fonction umask et vous essayez de l'utiliser pour restreindre les permissions pour vous-même. (EXPR & 0700).

**Unable to create sub named "%s"**

(F) Vous essayez de créer ou d'accéder à une fonction avec un nom illégal.

**Unbalanced context: %d more PUSHes than POPs**

(W) Le code de sortie a détecté un problème interne dans le nombre de contextes dans lequel il est entré et sorti.

**Unbalanced saves: %d more saves than restores**

(W) Le code de sortie a détecté un problème interne dans le nombre de valeurs qui ont été temporairement localisées.

**Unbalanced scopes: %d more ENTERs than LEAVEs**

(W) Le code de sortie a détecté un problème interne dans le nombre de blocs dans lequel il est entré et sorti.

**Unbalanced tmps: %d more allocs than frees**

(W)(W) Le code de sortie a détecté un problème interne dans le nombre de scalaires mortal qui ont été alloués et libérés.

**Undefined format "%s" called**

(F) Le format indiqué ne semble pas exister peut-être est-ce dans un autre paquetage ? Voir *perlform*.

**Undefined sort subroutine "%s" called**

(F) La fonction de comparaison de sort spécifiée ne semble pas exister. Peut-être est-ce dans un autre paquetage ? Voir `sort()` dans *perlfunc*.

**Undefined subroutine &%s called**

(F) La sous-fonction indiquée que vous essayez d'appeler n'a pas été définie, ou si elle l'a été, elle a été indéfinie depuis.

**Undefined subroutine called**

(F) La sous-fonction anonyme que vous essayez d'appeler n'a pas été définie, ou si elle l'a été, elle a été indéfinie depuis.

**Undefined subroutine in sort**

(F) La fonction de comparaison de sort spécifiée est déclarée mais ne semble pas avoir été définie pour le moment. Voir `sort()` dans *perlfunc*.

**Undefined top format "%s" called**

(F) Le format indiqué ne semble pas exister. Peut-être est-ce en réalité dans un autre paquetage ? Voir *perlform*.

**Undefined value assigned to typeglob**

(W) Une valeur indéfinie a été assignée à un typeglob, avec `*foo = undef`. Cela ne veut rien dire. Il est possible que vous pensiez en réalité à `undef *foo`.

**unexec of %s into %s failed!**

(F) La fonction `unexec()` a échoué pour une quelconque raison. Voir votre représentant FSF, qui vous a probablement mis cela en place le premier.

**Unknown BYTEORDER**

(F) Il n'y a pas de fonctions de swap de bits avec une machine avec cet ordre d'octets.

**unmatched () in regexp**

(F) Les parenthèses non précédées de backslash doivent toujours être équilibrées dans les expressions rationnelles. Si vous êtes utilisateur de vi, la touche % est utilisée pour trouver la parenthèse correspondante. Voir *perlre*.

**Unmatched right bracket**

(F) L'analyseur syntaxique a compté plus d'accolades fermées que d'ouvertes, donc vous avez probablement oublié d'en mettre une près de l'endroit que vous avez édité en dernier.

**unmatched [] in regexp**

(F) Les crochets autour d'une classe de caractères doivent se correspondre. Si vous voulez inclure un crochet fermant dans une classe de caractères, mettez la barre oblique inverse devant, ou mettez-la en premier. Voir *perlre*.

**Unquoted string "%s" may clash with future reserved word**

(W) Vous utilisez un simple mot qui peut être utilisé parfois comme mot réservé. Il est mieux de mettre un tel mot entre cotes, ou en lettres capitales, ou d'insérer un underscore dans son nom. Vous pouvez également le déclarer comme une sous-fonction.

**Unrecognized character %s**

(F) Le parseur Perl n'a aucune idée pour quoi faire avec le caractère spécifié dans votre script Perl (ou eval). Peut-être essayez-vous de lancer un script compressé, ou un programme binaire, ou un répertoire comme un programme Perl.

**Unrecognized signal name "%s"**

(F) Vous spécifiez un nom de signal à la fonction kill() qui n'est pas reconnue. Dites `kill -1` dans votre shell pour voir les signaux valides sur votre système.

**Unrecognized switch: -%s (-h will show valid options)**

(F) Vous avez spécifié une option illégale pour Perl. Ne faites pas cela. (Si vous ne pensez pas faire cela, vérifiez la ligne #! pour voir si vous n'avez pas spécifié la mauvaise option.)

**Unsuccessful %s on filename containing newline**

(W) Une opération de fichier a été tenté sur un nom de fichier, et cette opération a échoué, **PROBABLEMENT** parce que le nom de fichier contient un caractère de nouvelle ligne, **PROBABLEMENT** parce que vous avez oublié de faire un chop() ou un chomp(). Voir `chomp()` dans *perlfunc*.

**Unsupported directory function "%s" called**

(F) Votre machine ne supporte pas `opendir()` et `readdir()`.

**Unsupported function fork**

(F) Votre version d'exécutable ne supporte pas `fork()`.

Remarquez que sur certains systèmes, comme OS/2, il peut y avoir différentes versions de l'exécutable Perl, certains supportent `fork` d'autres non. Essayez de changer le nom que vous utilisez pour appeler Perl par `perl_`, à `perl_`, et ainsi de suite.

**Unsupported function %s**

(F) Votre machine n'implémente pas la fonction indiquée, apparemment. Ou du moins, Configure pense cela.

**Unsupported socket function "%s" called**

(F) Votre machine ne supporte pas le mécanisme des sockets Berkeley, Ou du moins, c'est ce que Configure pense.

**Unterminated <> operator**

(F) L'analyseur syntaxique voit un crochet gauche à la place de ce qu'il attendait être un terme, donc il regarde le crochet droit correspondant, et ne le trouve pas. Il y a des chances que vous ayez oublié une parenthèse obligatoire plus tôt dans la même ligne, et vous pensiez vraiment à un « moins que ».

**Use of "\$ \$ <digit>" to mean "\$ {\$ }<digit>" is deprecated**

(D) Les versions de Perl antérieures à la 5.004 interprétaient mal n'importe quel type marqué suivi par « \$ » et un chiffre. Par exemple « `$$0` » est compris incorrectement sous la forme « `$$0` » au lieu de « `${$0}` ». Ce bug est (la plupart du temps) corrigé dans Perl 5.004. Cependant, les développeurs de Perl 5.004 ne peuvent corriger ce bug complètement, car au moins 2 gros modules dépendent de cette ancienne façon de penser que « `$$0` » est une chaîne. Donc Perl 5.004 continue d'interpréter « `$$<digit>` » de la mauvaise façon dans les chaînes ; mais cela génère un message d'alerte. Et en Perl 5.005, ce traitement spécial cessera.

**Use of \$ # is deprecated**

(D) C'est une tentative échouée d'émuler les possibilités d'un pauvre **awk** Utilisez un explicite `printf()` ou `sprintf()` à la place.

**Use of \$ \* is deprecated**

(D) Cette variable magique qui rendait active la recherche d'un motif en multi-lignes, à la fois pour vous et pour les sous-fonctions chanceuses que vous serez amené à appeler. Vous devriez utiliser le nouveau `//m` et `//s` modificateurs pour faire cela sans prendre l'effet d'action-à-distance de `$*`.

**Use of %s in printf format not supported**

(F) Vous essayez d'utiliser une possibilité de `printf` qui est seulement accessible depuis le C. Cela veut dire habituellement qu'il existe une meilleure manière de faire ça en Perl.

**Use of bare << to mean <<" is deprecated**

(D) Vous êtes maintenant encouragé à utiliser l'explicite forme des cotes si vous envisagez d'utiliser une ligne vide comme séparateur d'un here-document.

**Use of implicit split to @\_ is deprecated**

(D) Cela fait beaucoup de travail pour le compilateur quand vous forcez la liste d'arguments d'une sous-fonction, donc il est mieux si vous affectez le résultat d'un split() explicitement dans un tableau (ou une liste).

**Use of inherited AUTOLOAD for non-method %s() is deprecated**

(D) Comme (ahem) dispositif accidentel, l' AUTOLOAD des sous-programmes sont recherchés comme des méthodes (utilisant la hiérarchie @ISA) même lorsque les sous-programmes à autoloader sont appelés en tant que tout simplement fonctions (par exemple Foo::bar()), pas comme méthodes (par exemple Foo->bar() ou \$obj->bar()). Cette anomalie sera rectifiée dans Perl 5,005, qui utilisera la consultation de méthode seulement pour l'AUTOLOADs des méthodes. Cependant, il y a une base significative de code existant qui peut utiliser le vieux comportement. Ainsi, comme étape d'intérim, Perl 5,004 émet un avertissement facultatif quand l'utilisation de non-méthodes a hérité de AUTOLOAD.

La règle simple est : l'héritage ne fonctionnera pas quand il y a autoloading de non-méthodes. La solution simple pour le vieux code est : Dans tout module qui dépendait d'hériter d'AUTOLOAD pour des non-méthodes d'une classe de base nommée BaseClass, exécutez \*AUTOLOAD = &BaseClass::AUTOLOAD pendant la mise en route.

Dans le code qui dit actuellement use AutoLoader; @ISA = qw(AutoLoader); vous pouvez ôter AutoLoader de @ISA et changer use AutoLoader; pour use AutoLoader 'AUTOLOAD';.

**Use of reserved word "%s" is deprecated**

(D) Le mot indiqué est un mot réservé. Les versions futures de Perl peuvent utiliser ce mot comme mot-clef, donc il se serait mieux de coter explicitement le mot d'une manière appropriée pour le contexte, ou d'utiliser un nom différent de toute façon. Ce message peut être supprimé pour les noms de fonction en ajoutant le préfixe &, ou en utilisant le qualifiant du paquetage, ex. &our(), ou Foo::our().

**Use of %s is deprecated**

(D) Le constructeur indique qu'il n'est plus recommandé de l'utiliser, généralement parce qu'il y a une meilleure façon de le faire, et parce que l'ancienne méthode a des mauvais effets de bord.

**Use of uninitialized value**

(W) Une valeur indéfinie a été utilisée comme si elle avait déjà été définie. Elle est interprétée comme un "<??>" ou 0, mais c'est peut-être un oubli. Pour supprimer ce message, donnez une valeur initiale à vos variables.

**Useless use of "re" pragma**

(W) Vous faites use re; sans aucun argument. Ce n'est pas vraiment utile.

**Useless use of %s in void context**

(W) Vous faites quelque chose dans effet de bord ??? dans un contexte qui ne fait rien avec la valeur retournée, comme un état qui ne retourne pas de valeur depuis le block, ou du côté gauche d'un opérateur virgule de scalaire. Très fréquemment ce point n'est pas stupide de votre part, mais c'est un échec de Perl pour parser le programme comme vous voudriez qu'il soit. Par exemple, vous obtenez ceci si vous mixez votre précedence C avec la précedence Python et dites ceci

```
$one, $two = 1, 2;
```

quand vous pensez dire

```
($one, $two) = (1, 2);
```

Une autre erreur commune est d'utiliser les parenthèses ordinaires pour construire une liste de références, lorsque vous pouvez utiliser les accolades ou les crochets, si vous dites

```
$array = (1,2);
```

alors que vous devriez dire

```
$array = [1,2];
```

Les crochets transforment explicitement une liste de valeurs en une valeur scalaire, alors que les parenthèses ne le font pas. Donc quand une liste entre parenthèses est évaluée dans un contexte de scalaire, la virgule est traitée comme l'opérateur C virgule, ce que qui ??? rejette l'argument de gauche, ce qui n'est pas ce que vous voulez. Voir *perlref* pour plus de détails sur ceci.

**untie attempted while %d inner references still exist**

(W) Une copie de l'objet retourné depuis tie (or tied) est toujours valide quand untie est appelé.

**Value of %s can be "0"; test with defined()**

(W) Dans une expression conditionnelle, vous utilisez <DESCRIPTEUR>, <\*> (glob), each(), ou readdir() comme une valeur booléenne. Chacun de ces constructeurs peut retourner une valeur de « 0 » ; Cela va rendre l'expression conditionnelle fausse, ce qui n'est probablement pas ce que vous désirez. Quand vous utilisez ces constructeurs dans une expression conditionnelle, testez leur valeur avec l'opérateur defined.

**Variable "%s" is not imported %s**

(F) Alors que « use strict » est en action, vous faites référence à une variable globale que vous pensez avoir importée d'un autre module, parce que quelque chose ayant le même nom (habituellement une sous-fonction) est exportée par ce module. Cela veut dire habituellement que vous avez mis un mauvais caractère devant votre variable.

**Variable "%s" may be unavailable**

(W) Une fonction interne *anonyme* est dans une fonction *nommée*, et en dehors de cela il y a une autre fonction ; et la fonction anonyme (la plus à l'intérieur) fait référence à une variable lexicale définie dans la fonction la plus à l'extérieur. Par exemple :

```
sub laplusalexterieur { my $a; sub aumilieu { sub { $a } } }
```

Si la sous-fonction anonyme est appelée ou référencée (directement ou indirectement) depuis la fonction la plus à l'extérieur, elle va partager la variable comme vous le souhaitez. Mais si la sous-fonction anonyme est appelée ou référencée quand la fonction la plus à l'extérieur est inactive, elle verra la valeur de la variable partagée comme elle était avant et durant le *\*premier\** appel à la fonction la plus à l'extérieur, ce qui n'est probablement pas ce que vous voulez.

Dans ces circonstances, il est habituellement mieux de faire la fonction du milieu anonyme, en utilisant la syntaxe sub {}. Perl a des supports spécifiques pour les variables partagées dans les fonctions anonymes internes ; une fonction nommée entre les deux interfère avec cette possibilité.

**Variable "%s" will not stay shared**

(W) Une sous-fonction interne *nommée* référence une variable lexicale définie dans une routine externe.

Quand la sous-fonction interne est appelée, elle va probablement voir la valeur de la variable de la fonction externe comme elle était avant et durant le *\*premier\** appel à la sous-fonction externe ; Dans ce cas, après que le premier appel à la sous-fonction externe soit fini, les sous-fonctions ne partageront plus une valeur commune pour la variable. En d'autres termes, la variable ne sera plus partagée.

De plus, si la sous-fonction externe est anonyme et référence une variable lexicale en dehors d'elle-même, alors les sous-fonctions interne et externe ne vont *jamais* partager la variable donnée.

Le problème peut habituellement être résolu en faisant la sous-fonction interne anonyme, en utilisant la syntaxe sub {}. Quand les fonctions internes qui référencent des variables dans les sous-fonctions externes sont appelées ou référencées, elles font automatiquement rebondir vers la valeur courante de chacune des variables.

**Variable syntax**

(A) Vous lancez accidentellement votre script par **csH** au lieu de Perl. Vérifiez la ligne avec #!, ou lancez votre script manuellement dans Perl.

**perl: warning: Setting locale failed.**

(S) Les message entiers ressemblent à ça :

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
 LC_ALL = "En_US",
 LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactement ce qui se passe lorsque le changement de configuration local échoue. Dans l'exemple, la configuration locale indique que LC\_ALL est à « En\_US » et que LANG n'a pas de valeur. Cette erreur indique que Perl a détecté que vous et/ou votre administrateur système ont configuré les soi-disant variables systèmes mais Perl ne peut utiliser cette configuration. Ce n'est pas très grave heureusement : Il existe une « default locale » appelée « C » que Perl peut et va utiliser, et le script va marcher. Mais avant que vous ne corrigiez ce problème, vous allez avoir le même message d'erreur à chaque fois que vous lancerez Perl. Comment vraiment corriger le problème est indiqué dans *perllocale* section **LOCALE PROBLEMS**.

**Warning: something's wrong**

(W) Vous passez à warn() une chaîne vide (l'équivalent d'un warn("")) ou vous l'appellez sans arguments et \$\_ est vide.

**Warning: unable to close filehandle %s properly**

(S) L'appel implicite à `close()` fait par `open()` donne une erreur indiquée dans le `close()`. Cela indique habituellement que votre système n'a plus d'espace disque.

**Warning: Use of "%s" without parentheses is ambiguous**

(S) Vous avez écrit un opérateur unaire suivi par quelque chose qui ressemble à un opérateur unaire qui peut être interprété comme un terme ou un opérateur unaire. Pour exemple, si vous savez que la fonction `rand` a un argument par défaut de 1.0, et que vous écrivez

```
rand + 5;
```

Vous PENSEZ que vous écrivez la même chose que

```
rand() + 5;
```

alors qu'en fait vous obtenez

```
rand(+5);
```

Donc mettez des parenthèses pour dire ce que vous pensez vraiment.

**Write on closed filehandle**

(W) Le descripteur de fichier que vous écrivez a été fermé quelque temps avant. Vérifiez votre flux logique de données.

**X outside of string**

(F) Vous avez un pack template qui spécifie une position relative avant le début de la chaîne qui est en train d'être décompactée. Voir `pack()` dans *perlfunc*.

**x outside of string**

(F) Vous avez un pack template qui spécifie une position relative après la fin de la chaîne qui est en train d'être décompactée. Voir `pack()` dans *perlfunc*.

**Xsub "%s" called in sort**

(F) L'utilisation d'un sous-programme comme comparaison avec `sort` n'est pas encore gérée.

**Xsub called in sort**

(F) L'utilisation d'un sous-programme comme comparaison avec `sort` n'est pas encore supporté.

**You can't use -1 on a filehandle**

(F) Un descripteur de fichier représente un fichier ouvert, et quand vous ouvrez le fichier, il a déjà passé tous les liens symboliques que vous présumiez essayer de regarder. Utilisez un nom de fichier plutôt.

**YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!**

(F) Et ce n'est probablement jamais ce que vous avez voulu, parce que vous n'avez pas les sources du kernel, et que votre vendeur ne vous donnera rien de ce que vous voulez. La meilleure solution est d'utiliser le script `wrapsuid` dans le répertoire pour mettre une couche C `setuid` autour de votre script.

**You need to quote "%s"**

(W) Vous avez affecté un mot comme nom d'un gestionnaire de signaux. Malheureusement, vous avez déjà un sous-programme avec ce nom, ce qui veut dire que Perl 5 va essayer ce sous-programme quand l'affectation va être exécutée. (Si c'est que vous voulez, mettre un `&` devant.)

**[gs]setsockopt() on closed fd**

(W) Vous essayez de lire ou de positionner une option sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de `socket()`? Voir `getsockopt()` dans *perlfunc*.

**\1 better written as \$ 1**

(W) En dehors des modèles ('patterns'), les références inversées fonctionnent comme des variables. L'utilisation de barres obliques inverses est historiquement du côté droit d'une substitution, mais stylistiquement???? il est mieux d'utiliser la forme variable car les autres programmeurs Perl s'y attendent, et cela marche mieux s'il existe plus de 9 références inversées.

**'|' and '<' may not both be specified on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commandes, et il a trouvé que l'entrée standard est un pipe, et que vous essayez de rediriger STDIN en utilisant '<'. Seulement un flux sur l'entrée standard pour un client, s'il vous plaît.

**'|' and '>' may not both be specified on command line**

(F) Une erreur spécifique à VMS. Perl fait ses propres redirections de ligne de commandes, et il pense que vous essayez de rediriger la sortie standard à la fois dans un fichier `and` dans un pipe vers une autre commande. Vous devez choisir l'un ou l'autre, en sachant que rien ne vous empêche de faire un pipe dans un programme ou dans un script Perl qui 'coupe' la sortie en deux flux, comme

```

open(OUT,">$ARGV[0]") or die "Can't write to $ARGV[0]: $!";
while (<STDIN>) {
 print;
 print OUT;
}
close OUT;

```

**Got an error from DosAllocMem**

(P) Une erreur spécifique à OS/2. Le plus probable est que vous utilisez une version obsolète de Perl, ce qui ne devrait pas arriver.

**Malformed PERLLIB\_PREFIX**

(F) Une erreur spécifique à OS/2. PERLLIB\_PREFIX doit être de la forme

```
prefix1;prefix2
```

ou

```
prefix1 prefix2
```

avec prefix1 et prefix2 non vides. Si prefix1 est un préfixe d'une librairie cherchée dans le path, prefix2 est substitué. L'erreur peut apparaître si un composant n'est pas trouvé, ou trop long. Voir « PERLLIB\_PREFIX » dans *README.os2*.

**PERL\_SH\_DIR too long**

(F) Erreur particulière à OS/2. PERL\_SH\_DIR est le répertoire où se trouve le shell sh-shell. Voir « PERL\_SH\_DIR » dans *README.os2*.

**Process terminated by SIG%s**

(W) C'est un message standard formulé par les applications OS/2, alors que les applications \*nix se terminent en silence. C'est considéré comme une caractéristique du portage sous OS/2. Ceci peut être facilement désactivé en positionnant le gestionnaire de signal approprié, voir Signaux in *perlipc*. Voir aussi « Process terminated by SIGTERM/SIGINT » dans *README.os2*.

## 35.2 TRADUCTION

### 35.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 35.2.2 Traducteurs

Alain Barbet <abarb@nmg.fr>

### 35.2.3 Relecture

Gérard et Inès Delafond.

# Chapitre 36

## perldebug

Débogage de Perl

### 36.1 DESCRIPTION

Tout d'abord, avez-vous essayé d'utiliser l'option **-w** ?

### 36.2 Le Débogueur Perl

Si vous invoquez Perl avec l'option **-d**, votre script tournera dans le débogueur de sources Perl. Il fonctionne comme un environnement Perl interactif, demandant des commandes de débogage qui vous laissent examiner le code source, placer des points d'arrêt, obtenir des traces des états passés de la pile, changer les valeurs des variables, etc. C'est si pratique que vous lancez souvent le débogueur tout seul juste pour tester interactivement des constructions en Perl afin de voir ce qu'elles font. Par exemple :

```
$ perl -d -e 42
```

En Perl, le débogueur n'est pas un programme séparé à la façon dont c'est habituellement le cas dans l'environnement compilé typique. À la place, l'option **-d** dit au compilateur d'insérer des informations sur le source dans les arbres d'analyse qu'il va donner à l'interpréteur. Cela signifie que votre code doit d'abord se compiler correctement pour que le débogueur travaille dessus. Puis lorsque l'interpréteur démarre, il précharge une bibliothèque Perl spéciale contenant le débogueur lui-même.

Le programme s'arrêtera *juste avant* la première instruction de son exécution (mais voyez plus bas en ce qui concerne les instructions pendant la compilation) et vous demandera d'entrer une commande de débogage. Contrairement à ce qu'on pourrait attendre, lorsque le débogueur s'arrête et vous montre une ligne de code, il affiche toujours la ligne qu'il est *sur le point* d'exécuter, plutôt que celle qu'il vient juste d'exécuter.

Toute commande non reconnue par le débogueur est directement exécutée (évaluée) comme du code Perl dans le paquetage courant (Le débogueur utilise le paquetage DB pour gérer les informations sur son propre état).

Tout espace blanc précédant ou suivant un texte entré au prompt du débogueur est d'abord supprimé avant tout autre traitement. Si une commande de débogage coïncide avec une fonction de votre propre programme, faites précéder simplement la fonction par quelque chose qui n'a pas l'air d'une commande de débogage, tel qu'un `;`, ou peut-être un `+`, ou en l'encadrant avec des parenthèses ou des accolades.

#### 36.2.1 Commandes du Débogueur

Le débogueur comprend les commandes suivantes :

##### **h** [commande]

Affiche un message d'aide.

Si vous fournissez une autre commande de débogage comme argument de la commande **h**, elle affichera uniquement la description de cette commande. L'argument spécial **h h** produira un listage d'aide plus compact, conçu pour tenir en un seul écran.

Si la sortie de la commande **h** (ou de toute commande, en fait) est plus longue que votre écran, faites-la précéder d'un symbole de tube pour qu'elle soit affichée page par page, comme dans



DB> |h

Vous pouvez changer le programme de pagination utilisé via la commande `O pager=...`

### **p expr**

Identique à `print {$DB:::OUT} expr` dans le paquetage courant. Cela signifie en particulier que, puisque c'est simplement la propre fonction `print` de Perl, les structures de données imbriquées et les objets ne sont pas affichés, contrairement à ce qui se passe avec la commande `x`.

Le handle de fichier `DB:::OUT` est ouvert vers `/dev/tty`, quelle que soit la redirection possible de `STDOUT`.

### **x expr**

Évalue son expression dans un contexte de liste et affiche le résultat d'une façon joliment formatée. Les structures de données imbriquées sont affichées récursivement, contrairement à ce qui se passe avec la vraie fonction `print`. Voir *Dumpvalue* si vous aimeriez faire cela vous-même.

Le format de sortie est gouverné par de multiples options décrites sous Options Configurables (§36.2.2).

### **V [pkg [vars]]**

Affiche la totalité (ou une partie) des variables du paquetage (avec par défaut `main`) en utilisant un joli afficheur de données (les hachages montrent leurs couples clé-valeur de façon que vous voyiez qui correspond à qui, les caractères de contrôle sont rendus affichables, etc.). Assurez-vous de ne pas placer là de spécificateur de type (comme `$`), mais juste les noms de symboles, comme ceci :

```
V DB filename line
```

Utilisez `~pattern` et `!pattern` pour avoir des expressions rationnelles positives et négatives.

Ceci est similaire au fait d'appeler la commande `x` pour chaque variable applicable.

### **X [vars]**

Identique à `V currentpackage [vars]`.

### **T**

Produit une trace de la pile. Voir plus bas pour des détails sur sa sortie.

### **s [expr]**

Pas à pas. Poursuit l'exécution jusqu'au début de l'instruction suivante, en descendant dans les appels de sous-programmes. Si une expression comprenant des appels de fonction est fournie, elle sera elle aussi suivie pas à pas.

### **n [expr]**

Suivant. Exécute les appels de sous-programme, jusqu'à atteindre le début de la prochaine instruction. Si une expression comprenant des appels de fonction est fournie, ces fonctions seront exécutées avec des arrêts avant chaque instruction.

### **r**

Continue jusqu'au retour du sous-programme courant. Affiche la valeur de retour si l'option `PrintRet` est mise (valeur par défaut).

### **<CR>**

Répète la dernière commande `n` ou `s`.

### **c [line|sub]**

Continue, en insérant facultativement un point d'arrêt valable une fois seulement à la ligne ou au sous-programme spécifié.

### **l**

Liste la prochaine fenêtre de lignes.

### **l min+incr**

Liste `incr+1` lignes en commençant à `min`.

### **l min-max**

Liste les lignes de `min` à `max`. `l -` est synonyme de `-`.

### **l line**

Liste une seule ligne.

### **l subname**

Liste la première fenêtre de lignes en provenance d'un sous-programme. *subname* peut être une variable contenant une référence de code.

-

Liste la précédente fenêtre de lignes.

**w [line]**

Liste une fenêtre (quelques lignes) autour de la ligne courante.

.

Retourne le pointeur de débogage interne sur la dernière ligne exécutée, et affiche cette ligne.

**f filename**

Passes à la visualisation d'un fichier différent ou d'une autre instruction `eval`. Si *filename* n'est pas un chemin complet tel que trouvé dans les valeurs de `%INC`, il est considéré être une expression rationnelle.

**/motif/**

Recherche un motif (une expression rationnelle de Perl) vers l'avant ; le / final est optionnel.

**?motif?**

Recherche un motif vers l'arrière ; le ? final est optionnel.

**L**

Liste tous les points d'arrêts et toutes les actions.

**S [![]regex]**

Liste les noms de sous-programmes [sauf] ceux correspondant à l'expression rationnelle.

**t**

Bascule le mode de traçage (voir aussi l'option `AutoTrace`).

**t expr**

Trace l'exécution de `expr`. Voir Exemples de Listages des Frames in *perldebguts* pour des exemples.

**b [ligne] [condition]**

Place un point d'arrêt avant la ligne donnée. Si ligne est omise, place un point d'arrêt sur la ligne qui est sur le point d'être exécutée. Si une condition est spécifiée, elle est évaluée chaque fois que l'instruction est atteinte : un point d'arrêt est réalisé seulement si la condition est vraie. Les points d'arrêt ne peuvent être placés que sur les lignes qui commencent une instruction exécutable. Les conditions n'utilisent pas `if` :

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

**b subname [condition]**

Place un point d'arrêt avant la première ligne du sous-programme nommé. *subname* peut être une variable contenant une référence de code (dans ce cas *condition* n'est pas supporté).

**b postpone subname [condition]**

Place un point d'arrêt à la première ligne du sous-programme après sa compilation.

**b load filename**

Place un point d'arrêt avant la première ligne exécutée du *fichier*, qui doit être un chemin complet trouvé parmi les valeurs `%INC`.

**b compile subname**

Place un point d'arrêt à la première instruction exécutée après que le sous-programme spécifié ait été compilé.

**d [ligne]**

Supprime un point d'arrêt sur la *ligne* spécifiée. Si *ligne* est omis, supprime le point d'arrêt sur la ligne sur le point d'être exécutée.

**D**

Supprime tous les points d'arrêt définis.

**a [ligne] commande**

Fixe une action devant être effectuée avant que la ligne ne soit exécutée. Si *ligne* est omis, place une action sur la ligne sur le point d'être exécutée. La séquence d'opérations réalisées par le débogueur est :

1. vérifie la présence d'un point d'arrêt sur cette ligne
2. affiche la ligne si nécessaire (trace)
3. effectue toutes les actions associées à cette ligne
4. interroge l'utilisateur en cas de point d'arrêt ou de pas à pas
5. évalue la ligne

Par exemple, ceci affichera \$foo chaque fois que la ligne 53 sera passée :

```
a 53 print "DB FOUND $foo\n"
```

#### **a [ligne]**

Supprime une action de la ligne spécifiée. Si *ligne* est omis, supprime l'action de la ligne sur le point d'être exécutée.

#### **A**

Supprime toutes les actions définies.

#### **W expression**

Ajoute une expression de surveillance (« watch » ? NDT) globale. Nous espérons que vous savez ce que c'est, car elles sont supposées être évidentes. **AVERTISSEMENT** : il est bien trop facile de détruire vos actions de surveillance en omettant accidentellement l'*expression*.

#### **W**

Supprime toutes les expressions de surveillance.

#### **O booloption ...**

Fixe chaque option booléenne listée à la valeur 1.

#### **O anyoption? ...**

Affiche la valeur d'une ou plusieurs options.

#### **O option=value ...**

Fixe la valeur d'une ou plusieurs options. Si la valeur contient des espaces, elle doit être mise entre guillemets. Par exemple, vous pouvez définir `O +pager="less -MQeicsNfr"` pour appeler **less** avec ces options spécifiques. Vous pouvez utiliser des apostrophes ou des guillemets, mais si vous le faites, vous devez protéger toutes les occurrences d'apostrophes ou de guillemets (respectivement) dans la valeur, ainsi que les échappements les précédant immédiatement mais ne devant pas les protéger. En d'autres termes, vous suivez les règles de guillemettage indépendamment du symbole ; e.g. : `O option='this isn\'t bad'` or `O option="She said, \"Isn't it?\""`. Pour des raisons historiques, le `=value` est optionnel, mais a pour valeur par défaut 1 uniquement lorsque cette valeur est sûre – c'est-à-dire principalement pour les options booléennes. Il vaut toujours mieux affecter une valeur spécifique en utilisant `=`. L'`option` peut être abrégé, mais ne devrait probablement pas l'être par souci de clarté. Plusieurs options peuvent être définies ensemble. Voir Options Configurables (§36.2.2) pour en trouver une liste.

#### **< ?**

Affiche toutes les commandes Perl constituant les actions précédant le prompt.

#### **< [ command ]**

Définit une action (une commande Perl) devant se produire avant chaque prompt du débogueur. Une commande sur plusieurs lignes peut être entrée en protégeant les fins de ligne par une barre oblique inverse. **AVERTISSEMENT** Si `command` est absent, toutes les actions sont effacées !

#### **<< command**

Ajoute une action (une commande Perl) devant se produire avant chaque prompt du débogueur. Une commande sur plusieurs lignes peut être entrée en protégeant les fins de ligne par une barre oblique inverse.

#### **> ?**

Affiche les commandes Perl constituant les actions suivant le prompt.

#### **> command**

Définit une action (une commande Perl) devant se produire après le prompt lorsque vous venez d'entrer une commande provoquant le retour à l'exécution du script. Une commande sur plusieurs lignes peut être entrée en protégeant les fins de ligne par des barres obliques inverses (parions que vous ne pouviez pas le deviner). **AVERTISSEMENT** Si `command` est absent, toutes les actions sont effacées !

#### **>> command**

Définit une action (une commande Perl) devant se produire après le prompt lorsque vous venez d'entrer une commande provoquant le retour à l'exécution du script. Une commande sur plusieurs lignes peut être entrée en protégeant les fins de ligne par des barres obliques inverses.

#### **{ ?**

Affiche les commandes du débogueur précédant le prompt.

#### **{ [ command ]**

Définit une action (une commande du débogueur) devant se produire avant chaque prompt du débogueur. Une commande sur plusieurs lignes peut être entrée de la façon habituelle. **AVERTISSEMENT** Si `command` est absent, toutes les actions sont effacées !

Puisque cette commande est en quelque sorte nouvelle, un avertissement est généré s'il apparaît que vous avez accidentellement entré un bloc. Si c'est ce que vous vouliez faire, écrivez-le sous la forme `{ ... }` ou même `do { ... }`.

**{{ command**

Ajoute une action (commande du débogueur) devant se produire avant chaque prompt du débogueur. Une commande sur plusieurs lignes peut être entrée, si vous ne savez pas comment : voyez ci-dessus.

**! number**

Relance une commande précédente (la commande précédente, par défaut).

**! -number**

Relance la nième commande précédente

**! pattern**

Relance la dernière commande ayant commencé par le motif. Voir aussi `0 recallCommand`.

**!! cmd**

Lance `cmd` dans un sous-processus (lisant sur `DB::IN`, écrivant sur `DB::OUT`). Voir aussi `0 shellBang`. Notez que le shell courant de l'utilisateur (en fait, sa variable `$ENV{SHELL}`) sera utilisé, ce qui peut interférer avec une interprétation correcte du statut ou du signal de sortie et les informations de `coredump`.

**H -number**

Affiche les `n` dernières commandes. Seules les commandes de plus de un caractère sont affichées. Si `number` est omis, les affiche toutes.

**q or ^D**

Quitte ("quit" ne marche pas ici, à moins que vous en ayez créé un alias). C'est la seule manière supportée pour quitter le débogueur, même si le fait d'entrer `exit` deux fois peut fonctionner.

Placez l'option `inhibit_exit` à 0 si vous voulez être en mesure de sauter jusqu'à la fin du script. Vous pouvez aussi avoir besoin de placer `$finished` à 0 si vous voulez avancer pas à pas dans la destruction globale.

**R**

Redémarre le débogueur en `exec()`utant une nouvelle session. Nous essayons de maintenir votre historique en cours de route, mais certains réglages internes et les options de la ligne de commande peuvent être perdus.

Les réglages suivants sont actuellement préservés : l'historique, les points d'arrêt, les actions, les options du débogueur, et les options de ligne de commande de Perl `-w`, `-I`, and `-e`.

**|dbcmd**

Exécute la commande du débogueur en redirigeant `DB::OUT` dans votre pager courant.

**||dbcmd**

De même que `|dbcmd` mais un `select` temporaire de `DB::OUT` est réalisé.

**= [alias value]**

Définit un alias de commande, comme

```
= quit q
```

ou liste les alias définis.

**command**

Exécute une commande en tant qu'instruction Perl. Un point-virgule final `y` sera ajouté. Si l'instruction Perl peut être confondue avec une commande du débogueur Perl, faites-la aussi précéder d'un point-virgule.

**m expr**

Liste quelles méthodes peuvent être appelées sur le résultat de l'expression évaluée. Cette évaluation peut être une référence à un objet consacré, ou à un nom de paquetage.

**man [manpage]**

Malgré son nom, cette commande appelle le visualisateur de documentation par défaut de votre système, pointant sur la page en argument, ou sur sa page par défaut si `manpage` est omis. Si ce visualisateur est **man**, les informations courantes de `Config` sont utilisées pour invoquer **man** via le `MANPATH` correct ou l'option `-M manpath`. Les recherches ratées de la forme `XXX` correspondant à des pages de manuel connues de la forme `perlXXX` seront réessayées. Ceci vous permet de taper `man debug` ou `man op` depuis le débogueur.

Sur les systèmes traditionnellement privés d'une commande **man** utilisable, le débogueur invoque **perldoc**. Cette détermination est occasionnellement incorrecte à cause de vendeurs récalcitrants ou, de façon plus fort à propos, du fait d'utilisateurs entreprenants. Si vous tombez dans une de ces catégories, fixez simplement manuellement la variable `$DB::doccmd` pour qu'elle pointe vers le visualisateur permettant de lire la documentation Perl sur votre système. Ceci peut être placé dans un fichier `rc`, ou défini via une affectation directe. Nous attendons toujours un exemple fonctionnel de quelque chose ressemblant à :

```
$DB::doccmd = 'netscape -remote http://something.here/';
```

### 36.2.2 Options Configurables

Le débogueur a de nombreuses options définissables via la commande `O`, soit de façon interactive, soit via l'environnement ou un fichier rc.

#### **recallCommand, ShellBang**

Les caractères utilisés pour rappeler une commande ou générer un nouveau shell. Par défaut, ces deux variables sont fixées à `!`, ce qui est malheureux.

#### **pager**

Programme à utiliser pour la sortie des commandes redirigées par un tube vers un paginateur (`pager ? NDT`) (celles qui commencent par un caractère `|`). Par défaut, `$ENV{PAGER}` sera utilisé. Puisque le débogueur utilise les caractéristiques de votre terminal courant pour les caractères gras et le soulignement, si le pager choisi ne transmet pas sans changements les séquences d'échappement, la sortie de certaines commandes du débogueur ne seront plus lisibles après qu'elles aient traversé le pager.

#### **tkRunning**

Exécute Tk pour le prompt (avec ReadLine).

#### **signalLevel, warnLevel, dieLevel**

Niveau de verbosité. Par défaut, le débogueur laisse tranquille vos exceptions et vos avertissements, parce que les altérer peut empêcher le bon fonctionnement de certains programmes. Il essaiera d'afficher un message lorsque des signaux INT, BUS ou SEGV arriveront sans être traités (mais voyez la mention des signaux dans BUGS (§36.6) ci-dessous).

Pour désactiver ce mode sûr par défaut, placez ces valeurs à quelque chose supérieur à 0. À un niveau de 1, vous obtenez une trace pour tous les types d'avertissements (c'est souvent gênant) et toutes les exceptions (c'est souvent pratique). Malheureusement, le débogueur ne peut pas discerner les exceptions fatales et non-fatales. Si `dieLevel` vaut 1, alors vos exceptions non-fatales sont aussi tracées et altérées sans cérémonie si elle proviennent de chaînes évaluées ou de tout type d'eval à l'intérieur des modules que vous essayez de charger. Si `dieLevel` est à 2, le débogueur ne se soucie pas de leur provenance : il usurpe vos handlers d'exceptions et affiche une trace, puis modifie toutes les exceptions avec ses propres embellissements. Ceci peut être utile pour certaines traces particulières, mais tend à désespérément détruire tout programme qui prend au sérieux sa gestion des exceptions.

#### **AutoTrace**

Mode de trace (similaire à la commande `t`, mais pouvant être mise dans `PERLDB_OPTS`).

#### **LineInfo**

Fichier ou tube dans lequel écrire les infos sur les numéros de ligne. Si c'est un tube (disons, `|visual_perl_db`), alors un court message est utilisé. C'est le mécanisme mis en oeuvre pour interagir avec un éditeur esclave ou un débogueur visuel, comme les hooks spéciaux de `vi` ou d'`emacs`, ou le débogueur graphique `ddd`.

#### **inhibit\_exit**

si à 0, permet *le passage direct* à la fin du script.

#### **PrintRet**

Affiche la valeur de retour après la commande `r` s'il est mis (par défaut).

#### **ornaments**

Affecte l'apparence de l'écran de la ligne de commande (voir `Term::ReadLine`). Il n'y a actuellement aucun moyen de les désactiver, ce qui peut rendre certaines sorties illisibles sur certains affichages, ou avec certains pagers. C'est considéré comme un bug.

#### **frame**

Affecte l'affichage des messages à l'entrée et à la sortie des sous-programmes. Si `frame & 2` est faux, les messages sont affichés uniquement lors de l'entrée (L'affichage à la sortie peut être utile si les messages sont entrecroisés).

Si `frame & 4` est faux, les arguments des fonctions sont affichés en plus du contexte et des infos sur l'appelant. Si `frame & 8` est faux, les FETCH surchargés chaînifiés (`stringify`) et liés (`tie`, `NDT`) sont autorisés sur les arguments affichés. Si `frame & 16` est faux, la valeur de retour du sous-programme est affichée.

La longueur à partir de laquelle la liste d'arguments est tronquée est régie par l'option suivante :

#### **maxTraceLen**

La longueur à laquelle la liste d'arguments est tronquée lorsque le bit 4 de l'option `frame` est mis.

Les options suivantes affectent ce qui se produit avec les commandes `V`, `X`, et `x` :

#### **arrayDepth, hashDepth**

Affiche seulement les N premiers éléments ("" pour tous).

**compactDump, veryCompact**

Change le style du vidage des tableaux et des hachages. Si `compactDump` est utilisé, les tableaux courts peuvent être affichés sur une seule ligne.

**globPrint**

Définit si l'on doit afficher le contenu des globalisations.

**DumpDBFiles**

Vidage des tableaux contenant des fichiers débogués.

**DumpPackages**

Vidage des tables de symboles des paquetages.

**DumpReused**

Vidage des contenus des adresses "réutilisées".

**quote, HighBit, undefPrint**

Change le style du vidage des chaînes. La valeur par défaut de `quote` est `auto` ; on peut permettre le vidage soit entre guillemets, soit entre apostrophes en la fixant à `"` ou `'` respectivement. Par défaut, les caractères dont le bit de poids fort est mis sont affichés *tels quels*.

**UsageOnly**

Vidage rudimentaire de l'usage de la mémoire par paquetage. Calcule la taille totale des chaînes trouvées dans les variables du paquetage. Ceci n'inclut pas les lexicaux dans la portée d'un fichier de module, ou perdus dans des fermetures.

Pendant le démarrage, les options sont initialisées à partir de `$ENV{PERLDB_OPTS}`. Vous pouvez y placer les options d'initialisation `TTY`, `noTTY`, `ReadLine`, et `NonStop`.

Si votre fichier `rc` contient :

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

alors votre script s'exécutera sans intervention humaine, plaçant les informations de trace dans le fichier `db.out` (Si vous l'interrompez, vous avez intérêt à réinitialiser `LineInfo` sur `/dev/tty` si vous voulez voir quelque chose).

**TTY**

Le TTY à utiliser pour les I/O de débogage.

**noTTY**

Si elle est mise, le débogueur entre en mode `NonStop`, sans se connecter à un TTY. En cas d'interruption (ou si le contrôle passe au débogueur via un réglage explicite de `$DB::signal` ou de `$DB::single` par le script Perl), il se connecte au TTY spécifié par l'option `TTY` au démarrage, ou à un TTY trouvé lors de l'exécution en utilisant le module `Term::Rendezvous` de votre choix.

Ce module doit implémenter une méthode appelée `new` qui retourne un objet contenant deux méthodes : `IN` et `OUT`. Celles-ci devraient retourner deux handles de fichiers à utiliser pour déboguer les entrées et sorties, respectivement. La méthode `new` devrait inspecter un argument contenant la valeur de `$ENV{PERLDB_NOTTY}` au démarrage, ou de `"/tmp/perlddbttty$$"` autrement. Ce fichier n'est pas inspecté du point de vue de la correction de son appartenance, des risques de sécurité sont donc théoriquement possibles.

**ReadLine**

Si elle est fautive, le support de `readline` dans le débogueur est désactivé de façon à pouvoir déboguer les application l'utilisant.

**NonStop**

Si elle est mise, le débogueur entre en mode non interactif jusqu'à ce qu'il soit interrompu manuellement, ou par programme en fixant `$DB::signal` ou `$DB::single`.

Voici un exemple de l'utilisation de la variable `$ENV{PERLDB_OPTS}` :

```
$ PERLDB_OPTS="NonStop frame=2" perl -d myprogram
```

Ceci exécutera le script `myprogram` sans intervention humaine, affichant l'arbre des appels avec leurs points d'entrée et de sortie. Notez que `NonStop=1 frame=2` est équivalent à `N f=2`, et qu'à l'origine, les options ne pouvait être abrégées que par leur première lettre (modulo les options `Dump*`). Il est néanmoins recommandé que vous les énonciez toujours complètement dans un souci de lisibilité et de compatibilité future.

D'autres exemples incluent

```
$ PERLDB_OPTS="NonStop frame=2" perl -d myprogram
```

qui exécute le script de façon non interactive, affichant les infos à chaque entrée dans un sous-programme et pour chaque ligne exécutée du fichier appelé *listing* (Si vous l'interrompez, vous feriez mieux de réinitialiser `LineInfo` vers quelque chose d'« interactif »!).

D'autres exemples incluent (en utilisant la syntaxe standard du shell pour montrer les valeurs des variables d'environnement) :

```
$ (PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out"
 perl -d myprogram)
```

qui peut être utile pour déboguer un programme qui utilise lui-même `Term::ReadLine`. N'oubliez pas de détacher le shell du TTY dans la fenêtre qui correspond à `/dev/ttyXX`, en entrant une commande comme, disons

```
$ sleep 1000000
```

Voir *Éléments Internes du Débugueur* in *perldebguts* pour plus de détails.

### 36.2.3 entrées/sorties du débogueur

#### Prompt

Le prompt du débogueur ressemble à

```
DB<8>
```

ou même

```
DB<<17>>
```

où ce nombre est le numéro de la commande, que vous utiliseriez pour y accéder avec le mécanisme intégré d'historique à la mode de **cs**. Par exemple, `!17` répéterait la commande numéro 17. La profondeur des signes inférieur à et supérieur à indique la profondeur d'imbrication du débogueur. Vous pouvez obtenir plus d'un ensemble de signes d'inégalité, par exemple, si vous êtes déjà sur un point d'arrêt et que vous affichez le résultat d'un appel de fonction qui lui-même contient un point d'arrêt, ou si vous sautez dans une expression via la commande `s/n/t` expression.

#### Commandes multilignes

Si vous désirez entrer une commande multiligne, telle qu'une définition de sous-programme contenant plusieurs instructions ou bien un format, protégez par une barre oblique inverse les fins de lignes qui termineraient normalement la commande de débogage. En voici un exemple :

```
DB<1> for (1..4) { \
cont: print "ok\n"; \
cont: }
ok
ok
ok
ok
```

Notez que cette affaire de protection d'une fin de ligne est spécifique aux commandes interactives tapées dans le débogueur.

#### Trace de la pile

Voici un exemple de ce dont pourrait avoir l'air une trace de la pile via la commande `T` :

```
$ = main::infested called from file 'Ambulation.pm' line 10
@ = Ambulation::legs(1, 2, 3, 4) called from file 'camel_flea' line 7
$ = main::pests('bactrian', 4) called from file 'camel_flea' line 4
```

Le caractère à gauche ci-dessus indique le contexte dans lequel la fonction a été appelée, avec `$` et `@` désignant les contextes scalaires et de liste respectivement, et `.` un contexte vide (qui est en fait une sorte de contexte scalaire). L'affichage ci-dessus signifie que vous étiez dans la fonction `main::infested` lorsque vous avez effectué le vidage de la pile, et qu'elle a été appelée dans un contexte scalaire à la ligne 10 du fichier *Ambulation.pm*, mais sans aucun argument, ce qui indique qu'elle a été appelée en tant que `&infested`. La ligne suivante de la pile montre que la fonction `Ambulation::legs` a été appelée dans un contexte de liste depuis le fichier *camel\_flea*, avec quatre arguments. La dernière ligne montre que `main::pests` a été appelée dans un contexte scalaire, elle aussi depuis *camel\_flea*, mais à la ligne 4.

Si vous exécutez la commande `T` depuis l'intérieur d'une instruction `use active`, la trace contiendra à la fois une ligne pour `require` et une ligne pour `eval`.

### Format de Listage des Lignes

Ceci montre les types de listage que la commande `l` peut produire :

```
DB<<13>> l
101: @i{@i} = ();
102:b @isa{@i,$pack} = ()
103 if(exists ${i{$prevpack}} || exists $isa{$pack});
104 }
105
106 next
107==> if(exists $isa{$pack});
108
109:a if ($extra-- > 0) {
110: %isa = ($pack,1);
```

Les lignes sur lesquelles on peut placer un point d'arrêt sont indiquées avec `:`. Les lignes ayant un point d'arrêt sont indiquées par `b` et celles ayant une actions par `a`. La ligne sur le point d'être exécutée est indiquée par `=>`.

### Listage de frame (par quoi traduire ? NDT)

Lorsque l'option `frame` est utilisée, le débogueur affiche les entrées dans les sous-programmes (et optionnellement les sorties) dans des styles différents. Voir *perldebguts* pour des exemples incroyablement longs de tout ceci.

## 36.2.4 Débogage des instructions lors de la compilation

Si vous avez des instructions exécutables lors de la compilation (comme du code contenu dans un bloc `BEGIN` ou `CHECK` ou une instruction `use`), elles ne seront pas stoppées par le débogueur, bien que les `requires` et les blocs `INIT` le soient, et les instructions de compilation peuvent être tracées avec l'option `AutoTrace` mise dans `PERLDB_OPTS`. Depuis votre propre code Perl, toutefois, vous pouvez transférer de nouveau le contrôle au débogueur en utilisant l'instruction suivante, qui est inoffensive si le débogueur n'est pas actif :

```
$DB::single = 1;
```

Si vous fixez `$DB::single` à 2, cela équivaut à avoir juste tapé la commande `n`, tandis qu'une valeur de 1 représente la commande `s`. La variable `$DB::trace` devrait être mise à 1 pour simuler le fait d'avoir tapée la commande `t`.

Une autre façon de déboguer le code exécutable lors de la compilation est de démarrer le débogueur, de placer un point d'arrêt sur le `load` d'un module :

```
DB<7> b load f:/perllib/lib/Carp.pm
Will stop on load of 'f:/perllib/lib/Carp.pm'.
```

puis de redémarrer le débogueur en utilisant la commande `R` (si possible). On peut utiliser `b compile subname` pour obtenir la même chose.

## 36.2.5 Personnalisation du Débogueur

Le débogueur contient probablement suffisamment de hooks de configuration pour que vous n'ayez jamais à le modifier vous-même. Vous pouvez changer le comportement du débogueur depuis le débogueur lui-même, en utilisant sa commande `O`, depuis la ligne de commande via la variable d'environnement `PERLDB_OPTS`, et par des fichiers de personnalisation.

Vous pouvez réaliser une certaine personnalisation en installant un fichier *.perldb* contenant du code d'initialisation. Par exemple, vous pourriez créer des alias ainsi (le dernier en est un que tout le monde s'attend à y voir) :

```
$DB::alias{'len'} = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'ps'} = 's/^ps\b/p scalar /';
$DB::alias{'quit'} = 's/^quit(\s*)/exit/';
```

Vous pouvez changer les options de *.perldb* en utilisant des appels comme celui-ci :

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```



Le code est exécuté dans le paquetage DB. Notez que *.perldb* est traité avant PERLDB\_OPTS. Si *.perldb* définit le sous-programme *afterinit*, cette fonction est appelée après que l'initialisation du débogueur soit terminée. *.perldb* peut être contenu dans le répertoire courant, ou dans le répertoire home. Puisque ce fichier est utilisé par Perl et peut contenir des commandes arbitraires, pour des raisons de sécurité, il doit être la propriété du superutilisateur ou de l'utilisateur courant, et modifiable par personne d'autre que son propriétaire.

Si vous voulez modifier le débogueur, copiez et renommez *perl5db.pl* dans la bibliothèque Perl et bidouillez-le à cœur joie. Vous voudrez ensuite fixer votre variable d'environnement PERL5DB pour qu'elle dise quelque chose comme ceci :

```
BEGIN { require "myperl5db.pl" }
```

En dernier recours, vous pouvez aussi utiliser PERL5DB pour personnaliser le débogueur en modifiant directement des variables internes ou en appelant des fonctions du débogueur.

Notez que toute variable ou fonction n'étant pas documentée ici (ou dans *perldebguts*) est considérée réservée à un usage interne uniquement, et est sujette en tant que telle à des modifications sans préavis.

### 36.2.6 Support de Readline

Tel que Perl est distribué, le seul historique de ligne de commande fourni est simpliste, vérifiant la présence de points d'exclamation en début de ligne. Toutefois, si vous installez les modules Term::ReadKey et Term::ReadLine du CPAN, vous aurez des possibilités d'édition comparables à celle que fournit le programme *readline(3)* du projet GNU. Recherchez-les dans le répertoire *modules/by-module/Term* du CPAN. Ceux-ci ne supportent toutefois pas l'édition normale de ligne de commande de *vi*.

Une complétion rudimentaire de la ligne de commande est aussi disponible. Malheureusement, les noms des variables lexicales ne sont pas disponibles dans la complétion.

### 36.2.7 Support d'un Éditeur pour le Débogage

Si la version d'**emacs** de la FSF est installée sur votre système, il peut interagir avec le débogueur Perl pour fournir un environnement de développement intégré cousin de ses interactions avec les débogueurs C.

Perl est aussi fourni avec un fichier de démarrage pour faire agir **emacs** comme un éditeur dirigé par la syntaxe comprenant (en partie) la syntaxe de Perl. Regardez dans le répertoire *emacs* de la distribution des sources de Perl.

Une configuration similaire de Tom Christiansen pour l'interaction avec toute version de *vi* et du X window system est aussi disponible. Celle-ci fonctionne de façon similaire au support multifenêtré intégré que fournit **emacs**, où c'est le débogueur qui dirige l'interface. Toutefois, à l'heure où ces lignes sont rédigées, la localisation finale de cet outil dans la distribution de Perl est encore incertaine.

Les utilisateurs de *vi* devraient aussi s'intéresser à **vim** et **gvim**, les versions pour rongeurs et huisseries, pour colorier les mots-clés de Perl.

Notez que seul perl peut vraiment analyser du Perl, ce qui fait que tous ces outils d'aide à l'ingénierie logicielle manquent un peu leur but, en particulier si vous ne programmez pas en Perl comme le ferait un programmeur C.

### 36.2.8 Le Profileur Perl

Si vous désirez fournir un débogueur alternatif à Perl, invoquez juste votre script avec un deux points et un argument de paquetage donné au drapeau **-d**. Un des débogueurs alternatifs les plus populaires pour Perl est le profileur Perl. Devel::Prof est maintenant inclus dans la distribution standard de Perl. Pour profiler votre programme Perl contenu dans le fichier *mycode.pl*, tapez juste :

```
$ perl -d:DProf mycode.pl
```

Lorsque le script se terminera, le profileur écrira les informations de profil dans un fichier appelé *tmon.out*. Un outil tel que **dproffp**, lui aussi fourni dans la distribution standard de Perl, peut être utilisé pour interpréter les informations de ce profil.

## 36.3 Débogage des expressions rationnelles

use `re 'debug'` vous permet de voir les détails sanglants sur la façon dont le moteur d'expressions rationnelles de Perl fonctionne. De façon à comprendre ces sorties typiquement volumineuses, on doit non seulement avoir une certaine idée de la façon dont les recherches d'expressions rationnelles fonctionnent en général, mais aussi connaître la façon dont Perl compile en interne ses expressions rationnelles pour en faire des automates. Ces sujets sont explorés en détails dans «Débogage des expressions rationnelles in *perldebugs*».

## 36.4 Débogage de l'usage de la mémoire

Perl contient un support interne du reporting de son propre usage de la mémoire, mais ceci est un concept plutôt avancé qui requiert une compréhension de la façon dont l'allocation mémoire fonctionne. Voir «Débogage de l'utilisation de la mémoire par Perl in *perldebugs*» pour les détails.

## 36.5 VOIR AUSSI

Vous avez essayé l'option `-w`, n'est-ce pas ?

Voir aussi *perldebugs*, *re*, *DB*, *Devel::Dprof*, *dprofpp*, *Dumpvalue* et *perlrun*.

## 36.6 BUGS

Vous ne pouvez pas obtenir d'informations concernant les frames de la pile ou de toute façon les fonctions de débogage qui n'ont pas été compilées par Perl, comme celles des extensions C ou C++.

Si vous modifiez vos arguments `@_` dans un sous-programme (comme avec `shift` ou `pop`), la trace de la pile ne vous en montrera pas les valeurs originales.

Le débogueur ne fonctionne actuellement pas en conjonction avec l'option de ligne de commande `-W`, puisqu'elle n'est elle-même pas exempte d'avertissements.

Si vous êtes dans un appel système lent (comme `wait`, `accept`, ou `read` depuis le clavier ou une socket) et que vous n'avez pas mis en place votre propre handler `$SIG{INT}`, alors vous ne pourrez pas revenir au débogueur par CTRL-C, parce que le propre handler `$SIG{INT}` du débogueur ne sait pas qu'il doit lever une exception pour faire un `longjmp(3)` hors d'un appel système lent.

## 36.7 TRADUCTION

### 36.7.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 36.7.2 Traducteur

Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>

### 36.7.3 Relecture

Gérard Delafond

# Chapitre 37

## perlvar

Variables prédéfinies en Perl

### 37.1 DESCRIPTION

#### 37.1.1 Noms prédéfinis

Les noms suivants ont une signification spéciale en Perl. La plupart de ces noms ont des mnémoniques acceptables ou équivalents dans l'un des shells. Néanmoins, si vous souhaitez utiliser des descripteurs longs, vous avez juste à ajouter

```
use English;
```

en tête de votre programme. Cela créera un alias entre les noms courts et les noms longs du module courant. Certains ont même des noms de longueur intermédiaire, généralement empruntés à **awk**. En général, il est préférable d'invoquer

```
use English '-no_match_vars';
```

si vous n'avez pas besoin de \$PREMATCH, \$MATCH, ou \$POSTMATCH, ce qui évite une baisse de performance certaine dans le traitement des expressions rationnelles. Voir *English*.

Les variables dépendant du descripteur de fichier courant peuvent être initialisées en appelant une méthode de l'objet IO::Handle, bien que ce soit moins efficace que d'utiliser les variables intégrées courantes. (Pour cela, les sous-titres ci-dessous contiennent le mot HANDLE). Vous devez écrire d'abord :

```
use IO::Handle;
```

après quoi vous pouvez utiliser soit

```
method HANDLE EXPR
```

soit, de manière plus sûre,

```
HANDLE->method(EXPR)
```

Chaque méthode retourne l'ancienne valeur de l'attribut IO::Handle. Les méthodes acceptent chacune EXPR en option, qui, s'il est précisé, spécifie la nouvelle valeur pour l'attribut IO::Handle en question. S'il n'est pas précisé, la plupart des méthodes ne modifient pas la valeur courante, exceptée autoflush(), qui fixera la valeur à 1, juste pour se distinguer.

Parce que le chargement de la classe IO::Handle est coûteux, vous devriez apprendre à utiliser les variables intégrées normales.

Quelques-unes de ces variables sont considérées comme étant en «lecture seule». Cela signifie que si vous essayez de leur attribuer une valeur, directement ou indirectement à travers une référence, vous obtiendrez une erreur d'exécution.

Vous devriez être très prudent quand vous modifiez les valeurs par défaut de la plupart des variables spéciales décrites dans ce document. Dans la plupart des cas vous devez localiser ces variables avant de les changer, puisque, si vous ne faites pas, le changement peut affecter d'autres modules qui comptent sur les valeurs par défaut des variables spéciales que vous avez changées. Voici une des façons correctes de lire un fichier entier en une seule fois :

```
open my $fh, "foo" or die $!;
local $/; # établit le mode slurp en local
my $content = <$fh>;
close $fh;
```

Mais le code suivant est assez mauvais :

```
open my $fh, "foo" or die $!;
undef $/; # établit le mode slurp
my $content = <$fh>;
close $fh;
```

puisque un autre module peut vouloir lire des données d'un fichier quelconque dans le «mode ligne» par défaut; donc si le code que nous venons juste de présenter a été exécuté, la valeur globale de `$/` est maintenant changée pour tout autre code qui tourne à l'intérieur du même interpréteur Perl.

Habituellement, quand une variable est localisée, vous voulez être sûr que ce changement ait la plus petite étendue possible. Donc, à moins que vous ne soyez déjà à l'intérieur d'un bloc court {}, vous devriez en créer un vous-même. Par exemple:

```
my $content = '';
open my $fh, "foo" or die $!;
{
 local $/;
 $content = <$fh>;
}
close $fh;
```

Voici un exemple de la manière dont votre propre code peut aller de travers :

```
for (1..5){
 nasty_break();
 print "$_ ";
}
sub nasty_break {
 $_ = 5;
 # do something with $_
}
```

Vous vous attendez probablement à ce que code affiche :

```
1 2 3 4 5
```

mais à la place vous obtenez:

```
5 5 5 5 5
```

Pourquoi? Parce que `nasty_break()` modifie `$_` sans d'abord le localiser. La solution est d'ajouter `local()` :

```
local $_ = 5;
```

C'est facile de repérer le problème dans un exemple court comme celui-là, mais dans un code plus compliqué vous cherchez à avoir des problèmes si vous ne localisez pas les changements des variables spéciales.

Dans la liste suivante, on trouvera les variables scalaires d'abord, puis les tableaux, et enfin les tableaux associatifs.

## \$ ARG

### \$ \_

La variable de stockage par défaut et l'espace de recherche de motif. Les paires suivantes sont équivalentes :

```
while (<>) {...} # équivalent seulement dans while!
while (defined($_ = <>)) {...}
```

```

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)

```

Voici les endroits où Perl utilisera `$_` même si vous ne le précisez pas :

- Diverses fonctions unaires, notamment les fonctions comme `ord()` et `int()`, ainsi que tous les tests sur les fichiers (`-f`, `-d`) à l'exception de `-t`, qui utilise par défaut STDIN.
- Diverses fonctions de liste comme `print()` et `unlink()`.
- Les opérations de recherche de motif `m//`, `s///`, et `tr///` quand elles sont utilisées sans l'opérateur `=`.
- La variable d'itération par défaut dans une boucle `foreach` si aucune autre variable n'est précisée.
- La variable implicite d'itération dans les fonctions `grep()` et `map()`.
- La variable par défaut où est stockée un enregistrement quand le résultat de `<FH>` s'autoteste et représente le critère unique d'un `while`. Attention : en dehors d'un test `while` cela ne marche pas.

(Moyen mnémorique: le *sous*-ligné est *sous*-entendu dans certaines opérations.)

## \$ a

## \$ b

Variables spéciales de paquetage quand `sort()` est utilisé, voyez `sort` in *perlfunc*. À cause de cette particularité `$a` et `$b` n'ont pas besoin d'être déclaré (en utilisant `use vars`, ou `our()`) même si vous utilisez le pragma `vars strict`. N'en faites pas des variables lexicales avec `my $a` ou `my $b` si vous voulez les utiliser dans un bloc ou une fonction de comparaison de `sort()`.

## \$ <chiffres>

Contient le groupement inclus dans les parenthèses correspondantes du dernier motif trouvé, sans prendre en compte les motifs trouvés dans les sous-blocs dont on est déjà sorti. (Mnémorique : comme `\<chiffres>`.) Ces variables sont en lecture seule et ont une portée dynamique étendue au BLOC courant.

## \$ MATCH

## \$ &

La chaîne de caractères trouvée par la dernière recherche de motif réussie (sans prendre en compte les motifs cachés dans un BLOC ou par un `eval()` inclus dans le BLOC courant). (Mnémorique : comme `&` dans certains éditeurs de texte.) Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir `BUGS`.

## \$ PREMATCH

## \$ ‘

La chaîne de caractères qui précède tout ce qui a été trouvé au cours de la dernière recherche de motif réussie (non comptées les correspondances cachées dans un BLOC ou un `eval()` du BLOC courant). (Mnémorique : `‘` est souvent utilisé, en anglais, comme guillemet ouvrant dans les citations). Variable en lecture seule.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir `BUGS`.

## \$ POSTMATCH

## \$ ’

La chaîne de caractères qui suit tout ce qui a été trouvé au cours de la dernière recherche de motif réussie (non comptées les correspondances cachées dans un BLOC ou un `eval()` du BLOC courant). (Mnémorique : `’` est souvent utilisé, en anglais, comme guillemet fermant dans les citations). Exemple :

```

local $_ = 'abcdefghi';
/def/;
print "$':&:$'\n"; # affiche abc:def:ghi

```

Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

L'utilisation de cette variable, n'importe où dans un programme, pénalise considérablement les performances de toutes les recherches de motifs. Voir `BUGS`.

## \$ LAST\_PAREN\_MATCH

**\$ +**

Le texte du dernier groupement parenthésé trouvé par une recherche de motif. Utile si vous ignorez lequel des motifs d'une alternative a donné un résultat. Exemple :

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnémonique: Soyez positif et allez de l'avant.) Variable en lecture seule dont la portée dynamique s'étend au BLOC courant.

**\$ ^N**

Le texte du groupement parenthésé le plus récemment fermé (c.-à-d. le groupe avec la parenthèse fermée la plus à droite) du modèle de la dernière recherche de motifs réussie. (Mnémonique: la parenthèse eNchâssée (éventuelle) qui a été plus récemment a fermé.)

Essentiellement utilisé à l'intérieur de blocs (`{...}`) pour examiner le texte correspondant le plus récent. Par exemple, pour capturer efficacement le texte dans une variable (en plus de `$1`, `$2`, etc.), remplacez (...) par

```
(?:...)(?{ $var = $^N })
```

Fixer et utiliser `$var` de cette manière vous évite d'avoir à vous soucier de savoir exactement combien il y a d'ensemble de parenthèses.

Variable dont la portée dynamique s'étend au BLOC courant.

**@LAST\_MATCH\_END****@+**

Ce tableau contient les positions [offsets] des fins des sous-chaînes correspondant aux groupements de la dernière recherche de motifs réussie dans la portée dynamique courante. `#[0]` est la position dans la chaîne de la fin de la recherche entière. C'est la même valeur que celle retournée par la fonction `pos` quand elle est appelée avec la variable sur laquelle porte la recherche. Le *n*-ième élément de ce tableau contient la position du *n*-ième groupement, donc `#[1]` est la position où `$1` fini, `#[2]` la position où `$2` fini, et ainsi de suite. Vous pouvez utiliser `#+` pour déterminer combien il y a de groupements dans la dernière recherche réussie. Voyez les exemples donnés pour la variable `@-`.

**\$ \***

Fixé à une valeur entière non nulle pour rechercher dans une chaîne multilignes, à 0 (ou `undef`) pour signifier à Perl que la chaîne ne contient qu'une seule ligne, dans le but d'optimiser la recherche. La recherche de motif sur des chaînes contenant plusieurs retours lignes peut produire des résultats étranges quand `"$"` est à 0 ou `undef`. La valeur par défaut est `undef`. (Mnémonique : \* remplace plusieurs.) Cette variable n'influence que l'interprétation de `"^"` et de `"$"`. Un retour ligne peut être recherché même si `$* == 0`.

L'utilisation de `"$"` est obsolète dans les Perl modernes, et est supplantée par les modificateurs de recherche `/s` et `/m`.

Assigner une valeur non numérique à `$*` déclenche un avertissement (et fait que `$*` agit comme si `$* == 0`), tandis que lui donner une valeur numérique fait qu'un `int` implicite est appliqué à cette valeur.

**HANDLE->input\_line\_number(EXPR)****\$ INPUT\_LINE\_NUMBER****\$ NR****\$ .**

Le numéro de la ligne courante du dernier descripteur de fichier auquel vous avez accédé.

Chaque descripteur de fichier en Perl compte le nombre de lignes qui ont été lues par son intermédiaire. (Dépendant de la valeur de `$/`, l'idée de ce que Perl ce fait d'une ligne peut ne pas correspondre à la votre.) Quand une ligne est lue d'un descripteur de fichier (via `readline()` or `<>`), ou quand `tell()` ou `seek()` est appelé sur lui, `$.` devient un alias du compteur de lignes pour ce descripteur de fichier.

Vous pouvez ajuster le compteur en assignant une valeur à `$.`, mais cela ne déplacera pas en fait le pointeur `seek`. *Localiser \$.* ne localisera pas le compteur du descripteur de fichier. Au lieu de cela, il localisera la notion que Perl a du descripteur de fichier pour lequel `$.` est un alias.

`$.` est réinitialisé quand le descripteur de fichier est fermé, mais **non** quand un descripteur de fichier ouvert est réouvert sans avoir été fermé par `close()`. Comme `<>` ne provoque pas de fermeture explicite, le numéro de ligne augmente au travers des fichiers ARGV (voir les exemples dans `eof` in *perlfunc*).

Vous pouvez aussi utiliser `HANDLE->input_line_number(EXPR)` pour avoir accès au compteur de lignes d'un descripteur de fichier donné sans avoir à vous tracasser de savoir quel est le dernier descripteur utilisé.

(Mnémonique : beaucoup de programmes utilisent "." pour pointer la ligne en cours)

**IO::Handle->input\_record\_separator(EXPR)**

**\$ INPUT\_RECORD\_SEPARATOR****\$ RS****\$ /**

Le séparateur d'enregistrement en lecture, par défaut retour-ligne. Il influence l'idée que Perl se fait d'une "ligne". Fonctionne comme la variable RS de **awk**, y compris le traitement des lignes vides comme des délimiteurs si initialisé à vide. (Note : une ligne vide ne peut contenir ni espace, ni tabulation.) Vous pouvez le définir comme une chaîne de caractères pour correspondre à un délimiteur multicaractère, ou à **undef** pour lire jusqu'à la fin du fichier. Notez que le fixer à `"\n\n"` est légèrement différent que de le fixer à `""` si le fichier contient plusieurs lignes vides consécutives. Le positionner à `""` traitera deux lignes vides consécutives (ou plus) comme une seule. Le positionner à `"\n\n"` implique que le prochain caractère lu appartient systématiquement à un nouveau paragraphe, même si c'est un retour-ligne. (Mnémonique: / est utilisé comme séparateur de ligne quand on cite une poésie.)

```
local $/; # activer le mode "slurp"
local $_ = <FH>; # Fichier complet depuis la position courante
s/\n[\t]+/ /g;
```

Attention : La valeur de `$/` doit être du texte et non une expression régulière. Il faut bien laisser quelque chose à **awk** :-)

Initialiser `$/` avec une référence à un entier, un scalaire contenant un entier, ou un scalaire convertissable en entier va provoquer la lecture d'enregistrements au lieu de lignes, avec une taille maximum par enregistrement correspondant à l'entier en référence. Donc :

```
local $/ = \32768; # ou \"32768", ou \${var_contenant_32768}
open my $fh, $myfile or die $!;
local $_ = <$fh>;
```

va lire un enregistrement d'une longueur maximale de 32768 octets depuis FILE. Si votre fichier ne contient pas d'enregistrements (ou si votre système d'exploitation ne supporte pas les fichiers d'enregistrements), vous obtiendrez probablement des valeurs incohérentes à chaque lecture. Si l'enregistrement est plus long que la taille spécifiée, il vous faudra le lire en plusieurs fois.

Sur VMS, les lectures d'enregistrements sont faites avec l'équivalent de **sysread**, donc il vaut mieux éviter de mélanger les lectures en mode enregistrements et en mode lignes sur le même fichier. (Cela n'est généralement pas un problème, puisque les fichiers que vous souhaiteriez lire en mode enregistrement sont probablement illisibles en mode ligne). Les systèmes non VMS effectuent des Entrées/Sorties standards, donc il est possible de mélanger les deux modes de lecture.

Voir aussi Les retours chariots in *perlport* et `$.`

**HANDLE->autoflush(EXPR)****\$ OUTPUT\_AUTOFLUSH****\$ |**

Si initialisé à une valeur différente de zéro, force une actualisation immédiatement et juste après chaque opération de lecture/écriture sur le canal de sortie sélectionné courant. La valeur par défaut est 0 (que le canal de sortie soit bufferisé par le système ou non; `$|` vous indique seulement si vous avez explicitement demandé à Perl d'actualiser après chaque écriture). Notez que **STDOUT** est typiquement bufferisé par ligne en sortie écran et par blocs sinon. Initialiser cette variable est surtout utile dans le cas d'une redirection de sortie, par exemple si vous exécutez un script Perl avec **rsh** et que vous voulez voir le résultat au fur et à mesure. Cela n'a aucun effet sur les buffers d'entrée. Voir *getc* in *perlfunc* pour cela. (Mnémonique : l'édition actualisée de vos redirections | )

**IO::Handle->output\_field\_separator EXPR****\$ OUTPUT\_FIELD\_SEPARATOR****\$ OFS****\$ ,**

Le séparateur de champs pour l'opérateur print. Si définie, cette valeur est affichée entre chacun des arguments de print. La valeur par défaut est **undef**. (Mnémonique : Ce qui est imprimé quand vous avez une "," dans vos champs)

**IO::Handle->output\_record\_separator EXPR****\$ OUTPUT\_RECORD\_SEPARATOR****\$ ORS****\$ \**

Le séparateur d'enregistrements pour l'opérateur print. Si définie, cette valeur est affichée après le dernier argument d'un print. (Mnémonique : positionnez "\$\" au lieu d'ajouter \n à la fin de chaque impression. Ou : Comme `$/`, mais c'est ce que vous "obtenez" de Perl.) (NdT. get **back** = "obtenez", `\` = **backslash**)

**\$ LIST\_SEPARATOR**

\$ "

Même chose que "\$," , sauf que cela s'applique aux tableaux de valeurs interpolées dans une chaîne de caractères entre guillemets doubles (ou toute chaîne interprétée d'une manière équivalente). La valeur par défaut est "espace". (Mnémonique : évident, je pense).

**\$ SUBSCRIPT\_SEPARATOR****\$ SUBSEP**

\$ ;

Le séparateur d'indices pour l'émulation de tableaux à plusieurs dimensions. Si vous vous référez à un élément de type tableau associatif comme

```
$foo{$a,$b,$c}
```

Cela signifie en fait

```
$foo{join($;, $a, $b, $c)}
```

Mais n'utilisez pas

```
@foo{$a,$b,$c} # une tranche -- notez le @
```

qui signifie

```
($foo{$a}, $foo{$b}, $foo{$c})
```

La valeur par défaut est "\034", la même que pour le SUBSEP en **awk**. Si vos clefs contiennent des valeurs binaires, il se peut qu'il n'y ait aucune valeur sûre pour "\$;". (Mnémonique : la virgule (le séparateur d'indices) est un demi-point-virgule. Ouais, je sais; c'est tiré par les cheveux, mais "\$," est déjà utilisé pour quelque chose de plus important.)

Envisagez d'utiliser de "vrais" tableaux à plusieurs dimensions comme décrit dans *perllo*.

\$ #

C'est le format de sortie des nombres. Cette variable est une pâle tentative d'émulation de la variable OFMT de **awk**. Il y a des cas, cependant, où Perl et **awk** ont des vues différentes sur la notion de numérique. La valeur par défaut est `%.ng`, où `n` est la valeur de la macro `DBL_DIG` du `float.h` de votre système. C'est différent de la valeur par défaut de OFMT en **awk** qui est `%.6g`, donc il vous faut initialiser `$#` explicitement pour obtenir la valeur **awk**. (Mnémonique : # est le signe des nombres – numéros)

L'utilisation de "\$#" est obsolète.

**HANDLE->format\_page\_number(EXPR)****\$ FORMAT\_PAGE\_NUMBER**

\$ %

Le numéro de la page en cours sur le canal de sortie en cours. Utilisés avec les formats. (Mnémonique : % est le numéro de page pour **nroff**.)

**HANDLE->format\_lines\_per\_page(EXPR)****\$ FORMAT\_LINES\_PER\_PAGE**

\$ =

La longueur de la page courante (en nombre de lignes imprimables) du canal de sortie en cours. Le défaut est 60. (Mnémonique : = est formé de lignes horizontales.)

**HANDLE->format\_lines\_left(EXPR)****\$ FORMAT\_LINES\_LEFT**

\$ -

Le nombre de lignes restantes sur la page en cours du canal de sortie courant. Utilisé avec les formats. (Mnémonique : lignes\_sur\_la\_page - lignes\_imprimées.)

**@LAST\_MATCH\_START**

@-

`$-[0]` est la position [offset] du début de la dernière recherche de motif réussie. `$-[n]` est la position du début de la sous-chaîne qui correspond au `n`-ième groupement, ou undef si le groupement ne correspond pas.

Donc, après une recherche de motif dans `$_`, `&` coïncide avec `substr $_, $-[0], $+[0] - $-[0]`. De la même manière, `$n` coïncide avec `substr $_, $-[n], $+[n] - $-[n]` si `$-[n]` est défini, et `$+` coïncide avec



`substr $_, $-[#-], $+[#-]`. On peut utiliser `$#-` pour trouver le dernier groupement dans la dernière recherche réussie. En opposition avec `$#+`, le nombre de groupements dans l'expression régulière. Comparez avec `@+`.

Ce tableau contient les positions des débuts des sous-chaînes correspondant aux groupements de la dernière recherche de motif réussie dans la portée dynamique courante. `$-[0]` est la position dans la chaîne du début de la chaîne trouvée entière. Le  $n$ -ième élément de ce tableau contient la position du  $n$ -ième groupement, donc `$+[1]` est la position où `$1` commence, `$+[2]` est la position où `$2` commence, et ainsi de suite.

Après une recherche sur une variable `$var` :

- `$'` est semblable à `substr($var, 0, $-[0])`
- `$&` est semblable à `substr($var, $-[0], $+[0] - $-[0])`
- `$'` est semblable à `substr($var, $+[0])`
- `$1` est semblable à `substr($var, $-[1], $+[1] - $-[1])`
- `$2` est semblable à `substr($var, $-[2], $+[2] - $-[2])`
- `$3` est semblable à `substr($var, $-[3], $+[3] - $-[3])`

#### **HANDLE->format\_name(EXPR)**

##### **\$ FORMAT\_NAME**

`$ ~`

Le nom du format de rapport courant pour le canal de sortie sélectionné. La valeur par défaut est le nom du descripteur de fichier. (Mnémonique : frère de "\$^".)

#### **HANDLE->format\_top\_name(EXPR)**

##### **\$ FORMAT\_TOP\_NAME**

`$ ^`

Nom du format de l'en-tête de page courant pour le canal de sortie sélectionné. La valeur par défaut est le nom du descripteur de fichier suivi de `_TOP`. (Mnémonique : pointe vers le haut de la page.)

#### **IO::Handle->format\_line\_break\_characters EXPR**

##### **\$ FORMAT\_LINE\_BREAK\_CHARACTERS**

`$ :`

L'ensemble des caractères courants après lesquels une chaîne de caractères peut être coupée pour passer à la ligne (commençant par `^`) dans une sortie formatée. La valeur par défaut est `"\n"`, pour couper sur les espaces ou les traits d'unions. (Mnémonique : en poésie un "deux-points" fait partie de la ligne.)

#### **IO::Handle->format\_formfeed EXPR**

##### **\$ FORMAT\_FORMFEED**

`$ ^L`

Ce que les formats utilisent pour passer à la page suivante. La valeur par défaut est `"\f"`.

#### **\$ ACCUMULATOR**

`$ ^A`

La valeur courante de la pile de la fonction `write()` pour formater les lignes avec `format()`. Un format contient des commandes `formline()`, qui stockent leurs résultats dans `$^A`. Après appel à son format, `write()` sort le contenu de `$^A` et le réinitialise. Donc vous ne voyez jamais le contenu de `$^A`, à moins que vous n'appeliez `formline()` vous-même, et ne regardiez le contenu. Voir *perform* et `formline()` in *perlfunc*.

#### **\$ CHILD\_ERROR**

`$ ?`

Le statut retourné par la dernière fermeture d'une redirection, une commande 'anti-apostrophe' (`"`), un appel réussi à `wait()` ou `waitpid()`, ou un opérateur `system()`. Notez que c'est le statut retourné par l'appel système `wait()` (ou cela lui ressemble). Le code de terminaison du sous-processus est (`$? >> 8`), et `$? & 127` indique quel signal (s'il y a lieu) a arrêté le processus, enfin `$? & 128` indique s'il y a eu un vidage mémoire [core dump]. (Mnémonique : similaire à **sh** et **ksh**.)

De plus, si la variable `h_errno` est supportée en C, sa valeur est retournée par `$?` si n'importe laquelle des fonctions `gethost*()` échoue.

Si vous avez installé un gestionnaire de signal pour `SIGHLD`, la valeur `$?` sera la plupart du temps erronée en dehors de ce gestionnaire.

A l'intérieur d'un sous-programme `END`, `$?` contient la valeur qui sera passée à `exit()`. Vous pouvez modifier `$?` dans un sous-programme `END` pour changer le code de sortie d'un script. Par exemple :

```

END {
 $? = 1 if $? == 255; # die lui donnerait la valeur 255
}

```

Sous VMS, la déclaration `use vmsish 'status'` fait renvoyer à `?$` le code de sortie réel de VMS, plutôt que le code d'émulation POSIX habituel `status`; voir `?$?` in *perl vms* pour les détails.

Voir aussi Indicateurs d'erreur.

## \$ {ENCODING}

La *référence d'objet* à l'objet `Encode` qui est utilisé pour convertir le code source en Unicode. Grâce à cette variable, votre script Perl n'a pas besoin d'être écrit en UTF-8. La valeur par défaut est *undef*. La manipulation directe de cette variable est fortement déconseillée. Voyez *encoding* pour plus de détails.

## \$ OS\_ERROR

## \$ ERRNO

## \$!

Dans un contexte numérique, contient la valeur courante de la variable `errno` ou, en d'autres termes, si un appel système ou à une bibliothèque échoue, il fixe cette variable. Cela signifie que la valeur de `$!` n'est significative qu'*immédiatement* après un **échec**.

```

if (open(FH, $filename)) {
 # Ici $! est sans signification.
 ...
} else {
 # Ici SEULEMENT $! est significative.
 ...
 # Ici, à nouveau, $! peut être sans signification.
}
Puisqu'ici nous pouvons avoir ou succès ou échec,
$! est sans signification ici.

```

Ci-dessus, *sans signification* est mis pour n'importe quoi : zéro, non-zéro, *undef*. Un appel réussi au système ou a une bibliothèque ne met pas la variable à zéro.

Dans un contexte textuel, contient le texte de l'erreur. Vous pouvez affecter un nombre à `$!` pour fixer `errno` si, par exemple, vous voulez utiliser `"$!"` pour retourner le texte de l'erreur *n*, ou si vous voulez fixer la valeur de l'opérateur `die()`. (Mnémonique : Plantage !)

Voir aussi Indicateurs d'erreur.

## %!

Chaque élément de `%!` à une valeur vraie seulement si `$!` est fixé à cette valeur. Par exemple, `!{ENOENT}` est vrai si et seulement si la valeur courante de `$!` est `ENOENT`; c'est-à-dire si la plus récente erreur a été "No such file or directory" (ou son équivalent moral : tous les systèmes d'exploitation ne donne pas exactement cette erreur et encore moins tous les langages). Pour vérifier si une clé particulière est significative sur votre système utilisez `exists $!{the_key}`; pour une liste des clés légales, utilisez `keys %!`. Voir *Errno* pour plus d'information, et voir aussi ci-dessus pour la validité de `$!`.

## \$ EXTENDED\_OS\_ERROR

## \$^E

Information d'erreur spécifique au système d'exploitation. Actuellement diffère de `$!` seulement sous VMS, OS/2 et Win32 (et MacPerl). Sur toutes les autres plates-formes, `$^E` est équivalent à `$!`.

Sous VMS, `$^E` fournit la valeur du statut VMS de la dernière erreur système. Les informations sont plus spécifiques que celles fournies par `$!`. Ceci est particulièrement important quand `$!` est fixé à **EVMSEERR**.

Sous OS/2, `$^E` correspond au code d'erreur du dernier appel API, soit au travers de CRT, soit directement depuis Perl.

Sous Win32, `$^E` retourne toujours l'information relative au dernier appel Win32 `GetLastError()`, qui décrit le dernier code d'erreur de l'API Win32. La plupart des applications spécifiques Win32 reportent les erreurs via `$^E`. ANSI C et UNIX positionnent `errno`, donc les programmes Perl les plus portables utiliseront `$!` pour remonter les messages d'erreur.

Les avertissements mentionnés dans la description de `$!` s'appliquent généralement à `$^E`. (Mnémonique : Extra-Explication d'Erreur)

Voir aussi Indicateurs d'erreur.

## \$ EVAL\_ERROR

**\$ @**

Le message d'erreur de syntaxe de la dernière commande `eval()`. Si `$@` est la chaîne nulle, le dernier `eval()` s'est exécuté correctement (bien que l'opération à exécuter ait pu échouer en mode normal). (Mnémonique : @ttention à l'erreur de syntaxe !)

Les messages d'avertissements ne sont pas récupérés dans cette variable. Vous pouvez toutefois construire une routine de traitement des avertissements en positionnant `$SIG{__WARN__}` comme décrit plus loin.

Voir aussi Indicateurs d'erreur.

**\$ PROCESS\_ID****\$ PID****\$ \$**

Numéro du processus Perl exécutant ce script. Vous devriez considérer cette variable comme étant en lecture seule, bien qu'elle puisse être modifiée à travers des appels à `fork()`. (Mnémonique : comme en shell.)

Note pour les utilisateurs de Linux : sur Linux, les fonctions C `getpid()` et `getppid()` retournent des valeurs différentes pour des fils d'exécution (threads) différents. Afin d'être portable, ce comportement n'est pas celui de `$ $` dont la valeur reste cohérente entre les fils d'exécution. Si vous souhaitez appeler la fonction `getpid()` réelle, vous pouvez utiliser le module CPAN `Linux::Pid`.

**\$ REAL\_USER\_ID****\$ UID****\$ <**

L'uid (id utilisateur) réel du processus. (Mnémonique : l'uid d'*OU* vous venez si vous exécutez `setuid()`.) Vous pouvez changer l'uid réel et l'uid effectif en même temps en utilisant `POSIX::setuid()`. Puisque une modification de `$<` nécessite un appel système, vérifiez `$!` après chaque tentative de modification pour détecter une éventuelle erreur.

**\$ EFFECTIVE\_USER\_ID****\$ EUID****\$ >**

L'uid effectif de ce processus. Exemple :

```
$< = $>; # Positionne l'uid réel à la valeur de l'uid effectif
(<,$>) = ($>,$<); # Inverse les uid réel et effectif
```

Vous pouvez à la fois changer l'uid réel et l'uid effectif en même temps en utilisant `POSIX::setuid()`.

(Mnémonique : l'uid *VERS* lequel vous alliez, si vous exécutez `setuid()`.) Note : `$<` and `$>` peuvent être inversés seulement sur les machines supportant `setreuid()`. Puisque une modification de `$>` nécessite un appel système, vérifiez `$!` après chaque tentative de modification pour détecter une éventuelle erreur.

**\$ REAL\_GROUP\_ID****\$ GID****\$ (**

Le gid (id de groupe) réel du processus. Si vous êtes sur une machine qui supporte l'appartenance simultanée à plusieurs groupes, renvoie une liste des groupes auxquels vous appartenez, séparés par des espaces. Le premier nombre retourné est le même que celui retourné par `getgid()`, les autres sont ceux retournés par `getgroups()`, avec possibilité de doublon entre l'un d'eux et le premier nombre.

Toutefois une valeur assignée à `$ (` doit être un nombre unique utilisé pour fixer le gid réel. Donc la valeur donnée par `$ (` ne doit *pas* être réassignée à `$ (` sans être forcée en numérique, par exemple en lui ajoutant 0.

Vous pouvez à la fois changer le gid réel et le gid effectif en même temps en utilisant `POSIX::setgid()`. Puisque une modification de `$ (` nécessite un appel système, vérifiez `$!` après chaque tentative de modification pour détecter une éventuelle erreur.

(Mnémonique : les parenthèses sont utilisées pour *GROUPER* les choses. Le gid réel est le groupe que vous laissez à votre *GAUCHE*, si vous utilisez `setgid()`.) (NdT: *GAUCHE* = parenthèse gauche)

**\$ EFFECTIVE\_GROUP\_ID****\$ EGID****\$ )**

Le gid effectif du processus. Si vous êtes sur une machine qui supporte l'appartenance simultanée à plusieurs groupes, renvoie une liste des groupes auxquels vous appartenez, séparés par des espaces. Le premier nombre retourné est le même que celui retourné par `getegid()`, les autres sont ceux retournés par `getgroups()`, avec possibilité de doublon entre l'un d'eux et le premier nombre.

De la même façon, une valeur assignée à `$)` doit être une liste de nombres séparés par des espaces. Le premier nombre est utilisé pour fixer le gid effectif, et les autres (si présents) sont passés à `setgroups()`. Pour obtenir le résultat d'une liste vide pour `setgroups()`, il suffit de répéter le nouveau gid effectif; c'est à dire pour forcer un gid effectif de 5 et un `setgroups()` vide il faut utiliser : `$) = "5 5" .`

Vous pouvez à la fois changer le gid réel et le gid effectif en même temps en utilisant `POSIX::setgid()` (utilise seulement un unique argument numérique). Puisque une modification de `$)` nécessite un appel système, vérifiez `#!` après chaque tentative de modification pour détecter une éventuelle erreur.

(Mnémonique : les parenthèses sont utilisées pour *GROUPER* les choses. Le gid effectif est le groupe qui vous donne le bon *DROIT* pour vous si vous exécutez `setgid()`.) (NdT: *DROIT* = parenthèse droite)

Note : `$<`, `$>`, `$(` et `$)` peuvent être positionnés seulement sur les machines qui supportent les fonctions correspondantes `set[re][ug]id()`. `$(` et `$)` peuvent être inversée seulement sur les machines supportant `setregid()`.

## \$ PROGRAM\_NAME

### \$ 0

Contient le nom du script Perl en cours d'exécution.

Sur certains systèmes (pas tous), assigner une valeur à `"$0"` modifie la zone d'argument que le programme `ps` voit. Sur certaines plateformes vous devez une option spéciale de `ps` ou un `ps` différent pour voir cette modification. C'est utile plutôt pour indiquer l'état courant du programme que pour cacher le programme en cours d'exécution. (Mnémonique : comme en `sh` et `ksh`.)

Notez qu'il existe une limite spécifique à chaque plateforme pour la longueur maximale de `$0`. Dans les cas les plus extrêmes, cette limite est la longueur de la valeur initiale de `$0`.

Sur certaines plateformes, il y a un remplissage arbitraire par, par exemple, des espaces après le nom modifié tel que vu par `ps`. Sur certaines plateformes, ce remplissage s'étend à la longueur initiale du nom quoique vous fassiez (c'est le cas de Linux 2.2 par exemple).

Note pour les utilisateurs de BSD : fixer la valeur de `$0` ne retire pas complètement "perl" de la sortie de `ps(1)`. Par exemple, donner la valeur "foobar" à `$0` peut donner le résultat "perl: foobar (perl)" (la présence ou l'absence du préfixe "perl:" ou du suffixe "(perl)" dépend de la variante et de la version BSD utilisée). C'est une caractéristique du système d'exploitation à laquelle Perl ne peut rien.

Dans des scripts multi-fils (multi-threads), chaque fil peut modifier sa copie de `$0` et Perl les coordonne afin que ce changement soit visible par `ps(1)` (en supposant que le système d'exploitation le permette). Notez que la valeur `$0` des autres fils n'est pas modifiée puisqu'ils en ont une copie qui leur est propre.

### \$ [

L'index du premier élément dans un tableau, et du premier caractère dans une sous-chaîne de caractère. La valeur par défaut est 0, mais vous pouvez la fixer à 1 pour faire ressembler Perl à `awk` (ou Fortran) quand vous utilisez les index ou les fonctions `index()` et `substr()`. (Mnémonique : `[` commence les index)

Depuis Perl 5, fixer la valeur de `"$["` est traité comme une directive de compilation et ne peut influencer le comportement des autres fichiers (ce qui explique que vous ne puissiez lui affecter qu'une valeurs constante à la compilation). Son usage est fortement déconseillé.

Notez que, contrairement à d'autres directives de compilation (tel que `strict`), la modification de `$[` est perçue dans tout le fichier, et donc en dehors de sa portée lexicale normale. En revanche, vous pouvez utiliser `local()` pour limiter sa portée à un bloc lexical.

### \$ ]

La version + le niveau de patch / 1000 de l'interpréteur Perl. Cette variable peut être utilisée pour déterminer si l'interpréteur Perl exécutant un script est au niveau de version souhaité. (Mnémonique : Cette version du Perl est-elle accrochée droite) (NdT : Les jeux de mots utilisés dans les mnémoniques anglais sont des plus délicats à rendre en français...) Exemple :

```
warn "No checksumming!\n" if $] < 3.019;
```

Voir également la documentation de `use VERSION` et `require VERSION` pour un moyen pratique d'empêcher l'exécution d'un script si l'interpréteur est trop vieux.

Lorsque vous testez cette variable, pour éviter les problèmes dus aux imprécisions numériques, utilisez les tests d'inégalité `<` et `>` plutôt que les tests contenant l'égalité comme `<=`, `==` et `>=`.

La représentation numérique en point flottant peut quelquefois amener des comparaisons numériques inexactes. Voir `$^V` pour une représentation plus moderne du numéro de la version Perl qui permet des comparaisons exactes sur les chaînes.

## \$ COMPILING

### \$ ^C

La valeur courante de l'indicateur binaire [flag] associé au commutateur **-c**. Principalement utilisé avec **-MO=...** pour permettre au code de modifier son comportement pendant qu'il est compilé, comme par exemple pour utiliser **AUTOLOAD** pendant la compilation plutôt que le chargement différé normal. Voir *See perlcc*. Fixer  $\$^C = 1$  est équivalent à appeler `B::minus_c`.

## \$ DEBUGGING

### \$ ^D

La valeur des options de débogage. (Mnémonique : valeur de l'option **-D**.) Peut être lue ou modifiée. Comme pour l'option équivalente en ligne de commande, vous pouvez utiliser soit une valeur numérique ( $\$^D = 10$ ) soit une valeur symbolique ( $\$^D = "st"$ ).

## \$ SYSTEM\_FD\_MAX

### \$ ^F

Le nombre maximum de descripteurs de fichier système, habituellement 2. Les descripteurs de fichiers système sont passés aux processus lancés par `exec()`, alors que ce n'est pas le cas pour les autres descripteurs de fichiers. De plus, pendant un `open()`; les descripteurs de fichiers système sont préservés même si `open()` échoue. (Les autres descripteurs sont fermés avant qu'un `open()` ne soient tenté.). Le statut fermeture-sur-exec d'un descripteur de fichier sera décidé suivant la valeur de  $\$^F$  au moment de l'ouverture du fichier, pipe ou socket correspondant et non au moment de l'`exec()`.

### \$ ^H

**ATTENTION** : Cette variable est disponible pour utilisation interne uniquement. Sa disponibilité, son comportement et son contenu sont soumis à changement sans avis.

Cette variable contient des directives de compilation pour l'interprète Perl. À la fin de la compilation d'un BLOC la valeur de cette variable est restaurée à la valeur qu'elle avait avant que l'interprète ne commence la compilation du BLOC.

Quand perl commence à analyser toute construction d'un bloc qui fournit une portée lexicale (par exemple, corps d'un eval, fichier requis, corps d'un sous-programme, corps de boucle, ou bloc conditionnel), la valeur existante de  $\$^H$  est sauvegardée, mais sa valeur est inchangée. Quand la compilation du bloc est terminée, elle reprend la valeur sauvegardée. Entre les points où sa valeur est sauvegardée et où elle est restaurée, le code des blocs **BEGIN** est libre de changer la valeur de  $\$^H$ .

Ce comportement fournit la sémantique du traitement de portée lexicale, et est utilisé, par exemple, dans le pragma `use strict`.

Le contenu devrait être un nombre entier; les différents bits de celui-ci sont utilisés comme indicateurs binaires de pragma. Voici un exemple:

```
sub add_100 { $^H |= 0x100 }

sub foo {
 BEGIN { add_100() }
 bar->baz($boon);
}
```

Considérons ce qui se passe pendant l'exécution du bloc **BEGIN**. À ce moment, le bloc **BEGIN** a déjà été compilé, mais le corps de `foo()` est encore à compiler. Par conséquent, la nouvelle valeur de  $\$^H$  ne sera visible seulement que pendant que le corps de `foo()` est compilé.

La substitution du bloc **BEGIN** précédent par :

```
BEGIN { require strict; strict->import('vars') }
```

montre comment `use strict 'vars'` est implémenté. Voici une version conditionnelle du même pragma lexical :

```
BEGIN { require strict; strict->import('vars') if $condition }
```

### % ^H

**ATTENTION** : Cette variable est disponible pour utilisation interne uniquement. Sa disponibilité, son comportement et son contenu sont soumis à changement sans avis.

Le hachage  $\%^H$  fournit la même sémantique du traitement de portée lexicale que  $\$^H$ . Ceci le rend utile pour implémenter des pragmas de portée lexicale.

## \$ INPLACE\_EDIT

### \$ ^I

La valeur courante de l'extension "édition sur place". Utilise `undef` pour mettre hors service l'édition sur place. (Mnémonique : valeur de l'option **-i**.)

**\$ ^M**

Par défaut, le dépassement de mémoire ne peut pas être capturé et entraîne une erreur fatale. Cependant, s'il est compilé pour cela, Perl peut utiliser le contenu de \$^M comme une réserve d'urgence après un die() dû à un dépassement de mémoire. Supposons que votre Perl ait été compilé avec l'option `-DPERL_EMERGENCY_SBRK` et utilise le malloc de Perl. Dans ce cas

```
$^M = 'a' x (1<<16);
```

allouera un buffer de 64K à utiliser en cas d'urgence. Voir le fichier *INSTALL* de votre distribution pour savoir comment ajouter vos propres options C de compilation lors de la compilation de perl. Pour décourager l'utilisation abusive de cette fonction avancée, il n'y a pas de noms longs English pour cette variable.

**\$ OSNAME****\$ ^O**

Le nom du système d'exploitation utilisé pour compiler cette copie de Perl, déterminé pendant le processus de configuration. Cette valeur est identique à `$Config{'osname'}`. Voir aussi *Config* et le commutateur en ligne de commande `-V` documenté dans *perlrun*.

Sur les plate-formes Windows \$^O n'est pas très utile: puisqu'elle donne toujours MSWin32, elle ne fait pas la différence entre 95/98/ME/NT/2000/XP/CE / .NET. Utilisez `Win32::GetOSName()` ou `Win32::GetOSVersion()` (voir *Win32* et *perlport*) pour distinguer ces variantes.

**\$ {^OPEN}**

Une variable interne utilisée par PerlIO. Une chaîne en deux parties, séparées par un octet `\0`, la première partie décrit les couches d'entrée, la seconde partie décrit les couches de sortie.

**\$ PERLDB****\$ ^P**

Variable interne pour le débogage. La signification des différents bits est sujet à changement, mais est actuellement :

1. x01  
Débugage d'un sous-programme enter/exit.
2. x02  
Débugage ligne à ligne.
3. x04  
Annuler les options d'optimisation.
4. x08  
Préserver plus de données pour les inspections interactives à venir.
5. x10  
Garder des informations sur les lignes sources où un sous-programme est défini.
6. x20  
Démarrer en mode pas à pas.
7. x40  
Dans le rapport, utiliser l'adresse du sous-programme au lieu de nom.
8. x80  
Mettre aussi les `goto &subroutine` dans le rapport.
9. x100  
Fournir des noms de fichier significatifs pour les evals, basés sur l'endroit où ils ont été compilés.
10. x200  
Fournir des noms significatifs pour les sous-programmes anonymes, basés sur l'endroit où ils ont été compilés.
11. x400  
Assertion de débogage de l'entrée/sortie des sous-programmes.

Note : Certains bits peuvent avoir un sens uniquement à la compilation, d'autres seulement à l'exécution. C'est un mécanisme nouveau, et les détails peuvent varier.

**\$ LAST\_REGEXP\_CODE\_RESULT****\$ ^R**

Résultat de l'évaluation de la dernière utilisation réussie d'une expression régulière (`{ code }`). Voir *perlre*. Cette variable est en lecture/écriture.

**\$ EXCEPTIONS\_BEING\_CAUGHT****\$ ^S**

État courant de l'interpréteur.

\$^S	État
undef	Analyse d'un module/eval
true (1)	En cours d'exécution d'un eval
false (0)	Autre

Le premier état peut apparaître dans un gestionnaire (handler) \$SIG{\_\_DIE\_\_} ou \$SIG{\_\_WARN\_\_}.

**\$ BASETIME****\$ ^T**

Heure à laquelle le script a démarré, en secondes depuis le 01/01/1970. Les valeurs retournées par les tests de fichiers **-M**, **-A**, et **-C** sont basées sur cette valeur.

**\$ {^TAINT}**

Indique si le mode souillé est actif ou non. 1 si actif (le programme a été lancé avec l'option **-T**), 0 si inactif, -1 si seuls les avertissements sont activés (avec **-t** ou **-TU**).

**\$ {^UNICODE}**

Indique l'état de certains réglage Unicode de Perl. Voir la documentation de l'option **-C** dans *perlrun* pour plus d'information sur les valeurs possibles. Cette variable est positionnée au lancement de Perl et n'est plus modifiable ensuite.

**\$ {^UTF8LOCALE}**

Cette variable indique si un locale UTF-8 a été détecté par perl au lancement. Cette information est utilisée par perl lorsqu'il est dans le mode d'ajustement de l'utf-8 au locale (lorsqu'on utilise l'option **-CL**); voir *perlrun* pour plus d'informations.

**\$ PERL\_VERSION****\$ ^V**

Les numéros de révision, version et sous-version de l'interpréteur Perl, sous la forme d'une chaîne composée des caractères portant ces numéros. Ainsi, dans Perl v5.6.0, elle est égale à `chr(5) . chr(6) . chr(0)` et `^V eq v5.6.0` retourne vrai. Notez que les caractères dans cette chaîne peuvent être potentiellement des caractères Unicode.

On peut l'utiliser pour déterminer si l'interpréteur Perl qui exécute un script est dans la bonne gamme de versions. (Mnémonique: utilisez `^V` pour contrôle de la Version.) Exemple:

```
warn "No \"our\" declarations!\n" if $^V and $^V lt v5.6.0;
```

Pour convertir `^V` en sa représentation en chaîne de caractères, utilisez le motif `%vd` de `sprintf()`:

```
printf "version is %vd\n", $^V; # Version de Perl
```

Voir la documentation de `use VERSION` et de `require VERSION` pour une manière commode de terminer le script si l'interpréteur Perl courant est trop ancien.

Voyez aussi `$]` pour une plus ancienne représentation de la version Perl.

**\$ WARNING****\$ ^W**

Valeur de l'option 'warning' (avertissement), initialement vrai si **-w** est utilisé, faux sinon, mais peut être directement modifié. (Mnémonique : comme l'option **-w**.) Voir aussi *warnings*.

**\$ {^WARNING\_BITS}**

L'ensemble des contrôles courants demandés avec le pragma `use warnings` Voir la documentation de `warnings` pour plus de détails.

**\$ EXECUTABLE\_NAME****\$ ^X**

Le nom utilisé pour exécuter la copie courante de Perl, vient de `argv[0]` en C ou de `/proc/self/exe` (lorsque ça existe).

Selon le système d'exploitation hôte, la valeur de `^X` peut être un chemin relatif ou absolu du fichier programme perl, ou peut être la chaîne utilisée pour invoquer perl et non le chemin du programme perl. Également, la plupart des systèmes d'exploitation permettent l'invocation de programmes qui ne sont pas dans la variable d'environnement

PATH, donc il n'y a aucune garantie que la valeur de \$^X soit dans PATH. Pour VMS, la valeur peut ou peut ne pas inclure de numéro de version.

Vous pouvez généralement utiliser la valeur de \$^X pour réinvoquer une copie indépendante du même perl qui est en train de tourner, par exemple :

```
@first_run = '$^X -le "print int rand 100 for 1..100"';
```

Mais rappelez-vous que tous les systèmes d'exploitation ne supportent pas le forking ou la capture de sortie des commandes, donc cette instruction complexe peut ne pas être portable.

Il n'est pas sans danger d'utiliser la valeur de \$^X comme un nom de chemin vers un fichier car certains systèmes d'exploitation qui ont un suffixe obligatoire pour les fichiers exécutables n'exige pas l'usage du suffixe quand on invoque une commande. Pour convertir la valeur de \$^X en un nom de chemin, utilisez les instructions suivantes:

```
Construit un ensemble de noms de fichier (pas des noms de commande).
use Config;
$this_perl = $^X;
if ($^O ne 'VMS')
 {$this_perl .= $Config{_exe}
 unless $this_perl =~ m/$Config{_exe}$/i;}
```

Parce que beaucoup de systèmes d'exploitation permettent à n'importe qui possédant un accès en lecture au fichier programme Perl d'en faire une copie, de modifier la copie, et ensuite d'exécuter cette copie, le programmeur Perl soucieux de sécurité doit prendre soin d'invoquer la copie installée de perl et non pas la copie référencée par \$^X. Les instructions suivantes permettent d'atteindre ce but, et produisent un chemin qui peut être invoqué comme une commande ou qui peut être référencé comme un fichier.

```
use Config;
$secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS')
 {$secure_perl_path .= $Config{_exe}
 unless $secure_perl_path =~ m/$Config{_exe}$/i;}
```

## ARGV

Le descripteur de fichier spécial qui itère sur les noms de fichier de la ligne de commande de @ARGV. Habituellement écrit comme le descripteur de fichier nul dans l'opérateur angle <>. Notez qu'actuellement ARGV n'a son effet magique qu'avec seulement l'opérateur <>; ailleurs c'est juste un descripteur de fichier ordinaire qui correspond au dernier fichier ouvert par <>. En particulier, passer \\*ARGV comme paramètre à une fonction qui attend un descripteur de fichier ne permet pas à cette fonction de lire automatiquement le contenu de tous les fichiers de @ARGV.

## \$ ARGV

Contient le nom du fichier courant quand on lit depuis <>.

## @ARGV

Contient les arguments de la ligne de commande du script. \$#ARGV est généralement le nombre d'argument moins 1 car \$ARGV[0] est le premier argument, et *non pas* le nom du script qui est dans "\$0". Voir "\$0" pour le nom du script.

## ARGVOUT

Le descripteur de fichier spécial qui pointe le fichier de sortie ouvert courant quand on fait de l'édition sur place avec I. Utile quand vous devez faire beaucoup d'insertions et que vous ne voulez pas modifier \$\_. Voir *perlrun* pour le commutateur I.

## @F

Le tableau @F contient les champs de chaque ligne lue quand le mode auto-découpage (autosplit) est activé. Voir *perlrun* pour le commutateur A. Ce tableau est spécifique au paquetage et doit être déclaré ou doit être invoqué avec un nom pleinement qualifié s'il n'est pas dans le paquetage main quand on utilise strict 'vars'.

## @INC

Contient la liste des répertoires où chercher pour évaluer les constructeurs `do EXPR`, `require`, ou `use`. Il est constitué au départ des arguments -I de la ligne de commande, suivi du chemin par défaut de la bibliothèque Perl, probablement */usr/local/lib/perl*, suivi de ".", pour représenter le répertoire courant. ( "." ne sera pas ajouté si le mode souillé (taint mode) est activé, que ce soit par -T ou par -t.) Si vous devez modifier cette variable à l'exécution vous pouvez utiliser `use lib` pour charger les bibliothèques dépendant de la machine :

```
use lib '/mypath/libdir/';
use SomeMod;
```



Vous pouvez aussi insérer des crochets [hooks] dans le système d'inclusion de fichiers en mettant du code Perl directement dans @INC. Ces crochets peuvent être des références à des sous-programmes, des références à des tableaux ou des objets bénis. Voir `require` in *perlfunc* pour les détails.

@\_

Dans un sous-programme, le tableau @\_ contient les paramètres passés à ce sous-programme. Voir *perlsub*.

%INC

Contient une entrée pour chacun des fichiers inclus par les opérateurs `do`, `require` ou `use`. La clef est le nom du fichier (avec les noms de modules convertis en chemin), et la valeur est la localisation du fichier effectivement trouvé. La commande `require` utilise ce hachage pour déterminer si un fichier donné a déjà été inclus.

Si le dossier a été chargé par un crochet (par exemple une référence à un sous-programme, voyez `require` in *perlfunc* pour une description de ces crochets), ce crochet est inséré par défaut dans %INC à la place d'un nom de fichier. Notez toutefois que le crochet a pu placer l'entrée dans %INC par lui-même pour fournir quelques infos plus spécifiques.

%ENV

\$ ENV{expr}

Contient les variables d'environnement. Fixer une valeur dans ENV change l'environnement pour les processus fils.

%SIG

\$ SIG{expr}

Contient les gestionnaires de divers signaux. Par exemple :

```
sub handler { # Le premier argument est le nom du signal
 my($sig) = @_;
 print "Caught a SIG$sig--shutting down\n";
 close(LOG);
 exit(0);
}

$SIG{'INT'} = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'} = 'DEFAULT'; # Restaure l'action par défaut
$SIG{'QUIT'} = 'IGNORE'; # ignore SIGQUIT
```

Donner la valeur 'IGNORE' a habituellement pour effet d'ignorer le signal, à l'exception du signal CHLD. Voir *perlipc* pour en savoir plus au sujet de ce cas spécial.

Voici d'autres exemples :

```
$SIG{"PIPE"} = "Plumber"; # suppose main::Plumber (Pas recommandé)
$SIG{"PIPE"} = \&Plumber; # Bien; suppose Plumber en sous-programme
$SIG{"PIPE"} = *Plumber; # quelque peu ésotérique
$SIG{"PIPE"} = Plumber(); # aïe, que retourne Plumber() ??
```

Assurez-vous de ne pas utiliser de mot nu comme nom de descripteur de signal, de peur que vous ne l'appeliez par inadvertance.

Si votre système reconnaît la fonction `sigaction()`, alors la gestion des signaux est implémentée par cette fonction. Ce qui signifie que vous avez une gestion des signaux fiable.

Depuis la version 5.8.0, par défaut, la réception d'un signal n'est plus immédiate (comprendre "non fiable") mais différée (comprendre "fiable"). Voir *perlipc* pour plus d'information.

On peut attacher un gestionnaire à certaines interruptions internes via la table de hachage %SIG. La routine indiquée par \$SIG{\_\_WARN\_\_} est appelée quand un message d'avertissement est sur le point d'être affiché. Le message est passé en premier argument. La présence de l'indicateur \_\_WARN\_\_ supprime les avertissements normaux sur STDERR. Vous pouvez vous en servir pour stocker les avertissements dans une variable, ou pour les transformer en erreurs fatales, comme ceci :

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

La routine indiquée par \$SIG{\_\_DIE\_\_} est appelée juste avant la gestion d'une erreur fatale, avec le message d'erreur comme premier argument. En sortie de la routine, le traitement de l'erreur reprend son cours normal, sauf si la routine provoque un arrêt du traitement via un `goto`, une sortie de boucle ou un `die()`. Le gestionnaire \_\_DIE\_\_

est explicitement désactivé pendant l'appel, de sorte que l'interruption `__DIE__` soit possible. Même chose pour `__WARN__`.

Dû à une erreur d'implémentation, le gestionnaire `$_SIG{__DIE__}` est appelé même à l'intérieur d'un `eval()`. N'utilisez pas ceci pour récrire une exception en suspens dans `$_@`, ou bizarrement pour surcharger `CORE::GLOBAL::die()`. Cette étrange action à distance devrait être supprimée dans une version future de manière que `$_SIG{__DIE__}` soit appelé uniquement quand votre programme est sur le point de se terminer, ce qui était l'intention à l'origine. Tout autre usage est désapprouvé.

Note : `__DIE__`/`__WARN__` ont ceci de spécial, qu'ils peuvent être appelés pour signaler une erreur (probable) pendant l'analyse du script. Dans ce cas, l'analyseur peut être dans un état instable, et toute tentative pour évaluer du code Perl provoquera certainement une faute de segmentation. Donc un appel entraînant une analyse du code devra être utilisée avec précaution, comme ceci :

```
require Carp if defined $^S;
Carp::confess("Un problème") if defined &Carp::confess;
die "Un problème, mais je ne peux charger Carp pour les détails...
 Essayer de relancer avec l'option -MCarp";
```

Dans cet exemple, la première ligne chargera `Carp` *sauf* si c'est l'analyseur qui a appelé l'interruption. La deuxième ligne affichera une trace de l'échec et arrêtera le programme si `Carp` est disponible. La troisième ligne ne sera exécutée que si `Carp` n'est pas disponible.

Voir `die` in *perlfunc*, `warn` in *perlfunc* et `eval` in *perlfunc* pour plus d'informations.

### 37.1.2 Indicateurs d'erreur

Les variables `$_@`, `$_!`, `$_E`, et `$_?` contiennent des informations à propos des différentes conditions d'erreur pouvant survenir pendant l'exécution d'un script Perl. Les variables sont données en fonction de la "distance" qui sépare le sous-système déclenchant l'erreur et le programme Perl. Elles correspondent respectivement aux erreurs détectées par l'interpréteur Perl, la bibliothèque C, le système d'exploitation ou un programme externe.

Pour illustrer ces différences, prenons l'exemple suivant :

```
eval q{
 open my $pipe, "/cdrom/install |" or die $!;
 my @res = <$pipe>;
 close $pipe or die "bad pipe: $?, $!";
};
```

Après l'exécution du code, les 4 variables peuvent être positionnées.

`$_@` le sera si l'expression à (eval)-uer ne s'est pas compilée (cela peut se produire si les fonctions `open` ou `close` ont été importées avec de mauvais prototypes), ou si le code Perl exécuté pendant l'évaluation échoue via `die()`. Dans ce cas `$_@` contient l'erreur de compilation, ou l'argument de `die` (qui interpolera `$_!` et `$_?`). (Néanmoins, voir aussi *Fatal*.)

Quand le code précédent est exécuté, `open()`, `<PIPE>`, et `close` sont traduits en appels à la librairie C et de là vers le noyau du système d'exploitation. La variable `$_!` est positionnée à la valeur `errno` de la bibliothèque C si l'un de ces appels échoue.

Sous quelques systèmes d'exploitation, `$_E` peut contenir un indicateur d'erreur plus verbeux, tel que, dans ce cas : "Le tiroir du CDROM n'est pas fermé." Sur les systèmes qui ne supportent pas de messages d'erreur étendus, `$_E` contient la même chose que `$_!`.

Finalement, `$_?` sera différent de `0` si le programme externe au script `/cdrom/install` échoue. Les huit bits de poids fort reflètent les conditions spécifiques de l'erreur rencontrée par le programme (la valeur `exit()` du programme). Les huit bits de poids faible reflètent le mode de l'échec, comme signal d'arrêt et information de vidage mémoire. Voir `wait(2)` pour les détails. Contrairement à `$_!` et `$_E` qui changent de valeur uniquement quand une erreur est détectée, la variable `$_?` est changée à chaque `wait` ou fermeture de pipe, écrasant l'ancienne valeur. C'est un peu comme `$_@` qui, sur chaque `eval()`, est toujours fixée en cas d'échec et vidée en cas de réussite.

Pour plus de détails, voir les descriptions individuelles de `$_@`, `$_!`, `$_E`, et `$_?`.

### 37.1.3 Note technique sur la syntaxe de noms variables

Les noms de variables en Perl peuvent avoir plusieurs formats. Habituellement, ils doivent commencer par une lettre ou un trait-de-soulignement, auquel cas ils peuvent être arbitrairement long (jusqu'à une limite interne de 251 caractères) et peuvent contenir des lettres, des chiffres, des traits-de-soulignement, ou la séquence spéciale `::` ou encore `'`. Dans ce cas, la partie placée avant le dernier `::` ou `'` est interprétée comme un *qualificateur de paquetage*; voir *perlmod*.

Les noms de variables en Perl peuvent être aussi une séquence de chiffres ou un signe de ponctuation unique ou un caractère de contrôle. Ces noms sont tous réservés par Perl pour des usages spéciaux ; par exemple, les noms `'tout en chiffres'` sont utilisés pour contenir les données capturées par les références-arrières d'une expression rationnelle après une recherche de motif réussie. Perl a une syntaxe spéciale pour les noms `'caractère de contrôle seul'` : il comprend `^X` (circonflexe X) comme signifiant le caractère contrôle-X. Par exemple, la notation `$^W` (dollar circonflexe W) est la variable scalaire dont le nom est le seul caractère contrôle-W. C'est mieux que de taper un contrôle-W directement dans votre programme.

Finalement, ce qui est nouveau dans Perl 5.6, les noms de variables peuvent être des chaînes alphanumériques qui commencent par des caractères de contrôle (ou mieux encore, un circonflexe). Ces variables doivent être écrites sous la forme `$_{Foo}`; les accolades ne sont pas facultatives. `$_{Foo}` dénote la variable scalaire dont le nom est un contrôle-F suivi par deux o. Ces variables sont réservées par Perl pour de futurs usages spéciaux, à l'exception de ceux qui commencent par `^_` (contrôle-soulignement ou circonflexe-soulignement). Aucun nom du type contrôle-caractère qui commence par `^_` n'acquerra de signification spéciale dans toute future version de Perl ; de tels noms peuvent donc être utilisés sans risque dans les programmes. Toutefois `$_` lui-même, *est* réservé.

Les identificateurs Perl qui commencent par des chiffres, des caractères de contrôle ou des caractères de ponctuation ne sont pas soumis aux effets de la déclaration `package` et sont toujours forcés dans le paquetage `main` ; ils ne sont pas non plus soumis aux erreurs de `strict 'vars'`. Quelques autres noms en sont aussi exemptés :

ENV	STDIN
INC	STDOUT
ARGV	STDERR
ARGVOUT	-
SIG	

En particulier, les nouvelles variables spéciales `$_{XYZ}` sont toujours associées au paquetage `main`, indépendamment de toute déclaration de paquetage dans la portée.

## 37.2 BUGS

Dû à un accident fâcheux dans l'implémentation de Perl, `use English` impose une pénalité de performance considérable sur toutes les recherches de motif dans un programme, indépendamment de l'endroit où elles se trouvent dans la portée de `use English`. Pour cette raison, l'utilisation de `use English` est fortement déconseillée dans les bibliothèques. Voir la documentation du module `Devel::SawAmpersand` du CPAN ( <http://www.cpan.org/modules/by-module/Devel/> ) pour plus d'information.

Avoir à penser à la variable `$_S` dans vos gestionnaires d'exceptions est tout simplement mauvais. `$_SIG{__DIE__}`, tel qu'il est implémenté actuellement, favorise le risque d'erreurs graves et difficiles à localiser. Évitez-le et utilisez un `END{}` ou `CORE::GLOBAL::die` à la place.

## 37.3 TRADUCTION

### 37.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 37.3.2 Traducteur

Fabien Martinet <ho.fmartinet@cma-cgm.com> (version de perl 5.00502). Mise à jour : Jean-Louis Morel <jl\_morel@bribes.org>, Paul Gaborit <paul.gaborit@enstimac.fr>.

### 37.3.3 Relecture

Jean-Louis Morel <jl\_morel@bribes.org>.

# Chapitre 38

## perlre

Les expressions rationnelles en Perl

### 38.1 DESCRIPTION

Cette page décrit la syntaxe des expressions rationnelles en Perl. (NdT: on emploie couramment le terme "expression régulière" car le terme anglais est "regular expression" qui s'abrège en "regex". Mais ne nous y trompons pas, en français, ce sont bien des "expressions rationnelles".)

Si vous n'avez jamais utilisé les expressions rationnelles, vous trouverez une rapide introduction dans *perlquick* et un tutoriel d'introduction un peu plus long dans *perlretut*.

Dans *Opérateurs d'expression rationnelle* in *perlop*, vous trouverez une présentation des opérateurs `m//`, `s///`, `qr//` et `??` avec une description de l'*usage* des expressions rationnelles dans des opérations de reconnaissances assortie de nombreux exemples.

Les opérations de reconnaissances peuvent utiliser différents modificateurs. Les modificateurs qui concernent l'interprétation des expressions rationnelles elles-mêmes sont présentés ici. Pour les modificateurs qui modifient l'usage des expressions rationnelles fait par Perl, regarder *Opérateurs d'expression rationnelle* in *perlop* et *Les détails sordides de l'interprétation des chaînes* in *perlop*.

**i**

Reconnaissance de motif indépendamment de la casse (majuscules/minuscules).

Si `use locale` est actif, la table des majuscules/minuscules est celle du locale courant. Voir *perllocale*.

**m**

Permet de traiter les chaînes multi-lignes. Les caractères `"^"` et `"$"` reconnaissent alors n'importe quel début ou fin de ligne plutôt qu'au début ou à la fin de la chaîne.

**s**

Permet de traiter une chaîne comme une seule ligne. Le caractère `"."` reconnaît alors n'importe quel caractère, même une fin de ligne qui normalement n'est pas reconnue.

Les modificateurs `/s` et `/m` passent outre le réglage de `$*`. C'est à dire que, quel que soit le contenu de `$*`, `/s` sans `/m` obligent `"^"` à reconnaître uniquement le début de la chaîne et `"$"` à reconnaître uniquement la fin de la chaîne (ou juste avant le retour à la ligne final). Combinés, par `/ms`, ils permettent à `"."` de reconnaître n'importe quel caractère tandis que `"^"` et `"$"` reconnaissent alors respectivement juste après ou juste avant un retour à la ligne dans la chaîne.

**x**

Augmente la lisibilité de vos motifs en autorisant les espaces et les commentaires.

Ils sont couramment nommés « le modificateur `/X` », même si le délimiteur en question n'est pas la barre oblique (`/`). En fait, tous ces modificateurs peuvent être inclus à l'intérieur de l'expression rationnelle elle-même en utilisant la nouvelle construction `(? . . .)`. Voir plus bas.

Le modificateur `/x` lui-même demande un peu plus d'explication. Il demande à l'interpréteur d'expressions rationnelles d'ignorer les espaces qui ne sont ni précédés d'une barre oblique inverse (`\`) ni à l'intérieur d'une classe de caractères. Vous pouvez l'utiliser pour découper votre expression rationnelle en parties (un peu) plus lisibles. Le caractère `#` est lui aussi traité comme un méta-caractère introduisant un commentaire exactement comme dans du code Perl ordinaire. Cela signifie que, si vous voulez de vrais espaces ou des `#` dans un motif (en dehors d'une classe de caractères qui n'est pas affectée par `/x`), vous devez les précéder d'un caractère d'échappement ou les coder en octal ou en hexadécimal. Prises ensembles, ces fonctionnalités rendent les expressions rationnelles de Perl plus lisibles. Faites attention à ne pas inclure le délimiteur de motif dans un commentaire (perl n'a aucun moyen de savoir que vous ne vouliez pas terminer le motif si tôt). Voir le code de suppression des commentaires `C` dans *perlop*.

### 38.1.1 Expressions rationnelles

Les motifs utilisés par la mise en correspondance de motifs sont des expressions rationnelles telles que fournies dans les routines de la Version 8 des expressions rationnelles. En fait, les routines proviennent (de manière éloignée) de la réécriture gratuitement redistribuable des routines de la Version 8 par Henry Spencer. Voir Version 8 des expressions rationnelles pour de plus amples informations.

Notamment, les méta-caractères suivants gardent leur sens à la *egrep* :

```

\ Annule le meta-sens du meta-caractère qui suit
^ Reconnaît le debut de la ligne
. Reconnaît n'importe quel caractère (sauf le caractère nouvelle ligne)
$ Reconnaît la fin de la ligne (ou juste avant le caractère nouvelle ligne final)
| Alternative
() Groupement
[] Classe de caractères

```

Par défaut, le caractère "^" ne peut reconnaître que le début de la ligne et le caractère "\$" que la fin (ou juste avant le caractère nouvelle ligne de la fin) et Perl effectue certaines optimisations en supposant que la chaîne ne contient qu'une seule ligne. Les caractères nouvelle ligne inclus ne seront donc pas reconnus par "^" ou "\$". Il est malgré tout possible de traiter une chaîne multi-lignes afin que "^" soit reconnu juste après n'importe quel caractère nouvelle ligne et "\$" juste avant. Pour ce faire, au prix d'un léger ralentissement, vous devez utiliser le modificateur /m dans l'opérateur de reconnaissance de motif. (Les anciens programmes obtenaient ce résultat en positionnant \$\* mais cette pratique est maintenant désapprouvée.)

Pour faciliter les substitutions multi-lignes, le méta-caractère "." ne reconnaît jamais un caractère nouvelle ligne à moins d'utiliser le modificateur /s qui demande à Perl de considérer la chaîne comme une seule ligne même si ce n'est pas le cas. Le modificateur /s passe outre le réglage de \$\* au cas où vous auriez quelques vieux codes qui le positionnerait dans un autre module.

Les quantificateurs standards suivants sont reconnus :

```

* Reconnaît 0 fois ou plus
+ Reconnaît 1 fois ou plus
? Reconnaît 0 ou 1 fois
{n} Reconnaît n fois exactement
{n,} Reconnaît au moins n fois
{n,m} Reconnaît au moins n fois mais pas plus de m fois

```

(Si une accolade apparaît dans n'importe quel autre contexte, elle est traitée comme un caractère normal. En particulier, la borne minimale n'est pas optionnelle) Le quantificateur "\*" est équivalent à {0,}, le quantificateur "+" à {1,} et le quantificateur "?" à {0,1}. n et m sont limités à des valeurs entières (!) inférieures à une valeur fixée lors de la compilation de perl. Habituellement cette valeur est 32766 sur la plupart des plates-formes. La limite réelle peut être trouvée dans le message d'erreur engendré par le code suivant :

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

Par défaut, un sous-motif quantifié est « gourmand », c'est à dire qu'il essaie de se reconnaître un maximum de fois (à partir d'un point de départ donné) sans empêcher la reconnaissance du reste du motif. Si vous voulez qu'il tente de se reconnaître un minimum de fois, il faut ajouter le caractère "?" juste après le quantificateur. Sa signification ne change pas. Seule sa « gourmandise » change :

```

*? Reconnaît 0 fois ou plus
+? Reconnaît 1 fois ou plus
?? Reconnaît 0 ou 1 fois
{n}? Reconnaît n fois exactement
{n,}? Reconnaît au moins n fois
{n,m}? Reconnaît au moins n fois mais pas plus de m fois

```

Puisque les motifs sont traités comme des chaînes entre guillemets, les séquences suivantes fonctionnent :

<code>\t</code>	tabulation	(HT, TAB)
<code>\n</code>	nouvelle ligne	(LF, NL)
<code>\r</code>	retour chariot	(CR)
<code>\f</code>	page suivante	(FF)
<code>\a</code>	alarme (bip)	(BEL)
<code>\e</code>	escape (pensez a troff)	(ESC)
<code>\033</code>	caractère en octal (pensez au PDP-11)	
<code>\x1B</code>	caractère hexadécimal	
<code>\x{263a}</code>	caractère hexadécimal étendu (Unicode SMILEY)	
<code>\c[</code>	caractère de contrôle	
<code>\N{nom}</code>	caractère nommé	
<code>\l</code>	convertit en minuscule le caractère suivant (pensez a vi)	
<code>\u</code>	convertit en majuscule le caractère suivant (pensez a vi)	
<code>\L</code>	convertit en minuscule jusqu'au prochain <code>\E</code> (pensez a vi)	
<code>\U</code>	convertit en majuscule jusqu'au prochain <code>\E</code> (pensez a vi)	
<code>\E</code>	fin de modification de casse (pensez a vi)	
<code>\Q</code>	désactive les meta-caractères de motif jusqu'au prochain <code>\E</code>	

Si `use locale` est actif, la table de majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est celle du locale courant. Voir *perllocale*. Pour de la documentation concernant `\N{nom}`, voir *charnames*.

Vous ne pouvez pas inclure littéralement les caractères `$` et `@` à l'intérieur d'une séquence `\Q`. Tels quels, ils se référeraient à la variable correspondante. Précédés d'un `\`, ils correspondraient à la chaîne `\$` ou `\@`. Vous êtes obligés d'écrire quelque chose comme `m/\Quser\E\@\Qhost/`.

Perl définit aussi les séquences suivantes :

<code>\w</code>	Reconnaît un caractère de "mot" (alphanumérique plus "_")
<code>\W</code>	Reconnaît un caractère de non "mot"
<code>\s</code>	Reconnaît un caractère d'espace.
<code>\S</code>	Reconnaît un caractère autre qu'un espace
<code>\d</code>	Reconnaît un chiffre
<code>\D</code>	Reconnaît un caractère autre qu'un chiffre
<code>\pP</code>	Reconnaît la propriété P (nommée). Utilisez <code>\p{Prop}</code> pour les noms longs
<code>\X</code>	Reconnaît une séquence d'un caractère Unicode étendue, équivalent à <code>(?:\PM\PM*)</code>
<code>\C</code>	Reconnaît un caractère (d'un seul octet) même sous <code>utf8</code> .

`\w` reconnaît un seul caractère alphanumérique (un caractère de l'alphabet ou un chiffre) ou `_`, pas à un mot entier. Pour reconnaître un mot entier au sens des identificateurs Perl, vous devez utiliser `\w+` (ce qui n'est pas la même chose que reconnaître les mots anglais ou français). Si `use locale` est actif, la liste de caractères alphanumériques couverts par `\w` dépend du locale courant. Voir *perllocale*. Vous pouvez utiliser `\w`, `\W`, `\s`, `\S`, `\d`, et `\D` à l'intérieur d'une classe de caractères mais si vous essayez des les utiliser comme borne d'un intervalle, ce n'est plus un intervalle et le "-" est compris littéralement. Si Unicode est actif, `\s` reconnaît aussi `"\x{85}"`, `"\x{2028}"` et `"\x{2029}"`. Voir *perlunicode* pour les détails à propos de `\pP`, `\PP` et `\X`, et *perluniintro* pour tout ce qui concerne Unicode en général. Vous pouvez définir de nouvelles propriétés pour `\p` et `\P`, voir *perlunicode*.

La syntaxe POSIX des classes de caractères :

```
[:class:]
```

est aussi disponible. Les classes disponibles et leur équivalent via la barre oblique inversée (si il existe) sont les suivants :

<code>alpha</code>	
<code>alnum</code>	
<code>ascii</code>	
<code>blank</code>	[1]
<code>cntrl</code>	
<code>digit</code>	<code>\d</code>
<code>graph</code>	
<code>lower</code>	

```

print
punct
space \s [2]
upper
word \w [3]
xdigit

```

**[1]**

Une extension GNU équivalente à [ \t ] (tous les espaces horizontaux).

**[2]**

Pas exactement équivalent à \s puisque [[:space:]] inclut aussi le (très rare) tabulateur vertical : "\ck" ou chr(11).

**[3]**

Une extension Perl, voir plus loin.

Par exemple, utilisez [[:upper:]] pour reconnaître tous les caractères minuscules. Remarquez que les crochets ([]) font partie de la construction [[:]] et non de la classe de caractères. Par exemple :

```
[01[:alpha:]]%
```

reconnaît les chiffres un et deux, le caractère pourcent et tous les caractères alphabétiques.

Voici un tableau d'équivalence des classes avec les constructions Unicode \p{} et avec les classes représentées par un caractère précédé d'un backslash lorsqu'elles existent :

[[:...:]]	\p{...}	backslash
alpha	IsAlpha	
alnum	IsAlnum	
ascii	IsASCII	
blank	IsSpace	
cntrl	IsCntrl	
digit	IsDigit	\d
graph	IsGraph	
lower	IsLower	
print	IsPrint	
punct	IsPunct	
space	IsSpace	
	IsSpacePerl	\s
upper	IsUpper	
word	IsWord	
xdigit	IsXDigit	

Par exemple [[:lower:]] et \p{IsLower} sont équivalents.

Si le pragma utf8 n'est pas actif mais que le pragma locale l'est, les classes sont corrélées via l'interface isalpha(3) (sauf pour "word" et "blank").

Les classes nommées non évidentes sont :

**cntrl**

N'importe quel caractère de contrôle. Ce sont habituellement des caractères qui ne produisent aucune sortie mais qui, par contre, modifie le terminal : par exemple newline (passage à la ligne) et backspace (retour arrière) sont des caractères de contrôle. Tous les caractères pour lesquels ord() renvoie une valeur inférieure à 32 sont des caractères de contrôle (avec l'ASCII, les codages de caractères ISO Latin et Unicode) ainsi que le caractère dont la valeur ord() vaut 127 (DEL).

**graph graph**

N'importe quel caractère alphanumérique ou de ponctuation (ou spécial).

**print**

N'importe quel caractère alphanumérique, de ponctuation (ou spécial) ou espace.

**punct**

N'importe quel caractère de ponctuation (ou spécial).

**xdigit**

Un chiffre hexadécimal. Bien que peu utile ([0-9A-Fa-f] fonctionne très bien), elle est incluse pour la complétude.

Vous pouvez utiliser la négation d'une classe de caractères [[:]] en préfixant son nom par un '^'. C'est une extension de Perl. Par exemple :

POSIX	traditionnel	Unicode
[:^digit:]	\D	\P{IsDigit}
[:^space:]	\S	\P{IsSpace}
[:^word:]	\W	\P{IsWord}

Perl respecte le standard POSIX puisque les classes de caractères POSIX ne sont supportées qu'à l'intérieur d'une classe de caractères. Les deux classes de caractères POSIX [.cc.] et [=cc=] sont reconnues **sans** être supportées : toute tentative d'utilisation de ces classes provoquera une erreur.

Perl définit les assertions de longueur nulle suivantes :

```
\b Reconnait la limite d'un mot
\B Reconnait autre chose qu'une limite de mot
\A Reconnait uniquement le debut de la chaîne
\Z Reconnait uniquement la fin de la chaîne (ou juste avant le
 caractère de nouvelle ligne final)
\z Reconnait uniquement la fin de la chaîne
\G Reconnait l'endroit ou s'est arrete le precedent m//g
 (ne fonctionne qu'avec /g)
```

Une limite de mot (\b) est définie comme le point entre deux caractères qui sont d'un côté un \w et de l'autre un \W (dans n'importe quel ordre). Le début et la fin de la chaîne correspondent à des caractères imaginaires de type \W. (À l'intérieur d'une classe de caractères, \b représente le caractère "backspace" au lieu d'une limite de mot.) \A et \Z agissent exactement comme "" et "\$" sauf qu'ils ne reconnaissent pas les lignes multiples quand le modificateur /m est utilisé alors que "" et "\$" reconnaissent toutes les limites de lignes internes. Pour reconnaître la fin réelle de la chaîne, en tenant compte du caractère nouvelle ligne, vous devez utiliser \z.

\G est utilisé pour enchaîner plusieurs mises en correspondance (en utilisant m//g). Voir Opérateurs d'expression rationnelle in *perlop*. Il est aussi utile lorsque vous écrivez un analyseur lexicographique à la *lex* ou lorsque vous avez plusieurs motifs qui doivent reconnaître des sous-chaînes successives de votre chaîne. Voir la référence précédente. L'endroit réel à partir duquel \G va être reconnu peut être modifié en affectant une nouvelle valeur à pos(). Voir pos in *perlfunc*. Actuellement, \G n'est utilisable que lorsqu'il est placé en tout début de motif; bien qu'utilisable partout en théorie, comme dans /(?<=\G. .) ./g, quelques cas (comme, par exemple, /.\G/g) posent problème et il est donc recommandé de ne pas le faire.

La construction (. . .) crée des zones de mémorisation (des sous-motifs). Pour vous référer au *n*-ième sous-motifs, utilisez \n à l'intérieur du motif. À l'extérieur du motif, utilisez toujours "\$" à la place de "\" devant *n*. (Bien que la notation \n fonctionne en de rares occasions à l'extérieur du motif courant, vous ne devriez pas compter dessus. Voir l'avertissement au sujet de \1 et de \$1.) Une référence à un sous-motif mémorisé est appelée une référence arrière.

Vous pouvez utiliser autant de sous-motifs que nécessaire. Par contre, Perl utilise les séquences \10, \11, etc. comme des synonymes de \010, \011, etc. (Souvenez-vous que 0 signifie octal et donc \011 est le caractère codé par un neuf dans votre jeu de caractère, une tabulation en ASCII.) Perl résout cette ambiguïté en interprétant \10 comme une référence arrière uniquement si il y a déjà au moins 10 parenthèses ouvrantes avant. De même, \11 sera une référence arrière uniquement si il y a au moins onze parenthèses ouvrantes avant. Et ainsi de suite. Les séquences de \1 à \9 sont toujours interprétées comme des références arrières.

Exemples :

```
s/^([^]*) *([^]*)/$2 $1/; # echange les deux premiers mots

if (/(\.)\1/) { # trouve le premier caractère répété
 print "'$1' is the first doubled character\n";
}
```



```
if (/Time: (..):(..):(..)/) {
 $hours = $1;
 $minutes = $2;
 $seconds = $3;
}
```

Plusieurs variables spéciales se réfèrent à des portions de la dernière reconnaissance. `$+` renvoie le dernier sous-motif entre parenthèses reconnu. `&` renvoie le dernier motif reconnu. (Autrefois `$0` était utilisé pour le même usage mais maintenant, il renvoie le nom du programme.) `$'` renvoie tout ce qui est avant le motif reconnu. `$'` renvoie tout ce qui est après le motif reconnu. Et `$^N` renvoie ce qui a été reconnu par le dernier sous-motif refermé. `$^N` peut-être utilisé dans les motifs étendus (voir plus bas), par exemple pour affecter un sous-motif à une variable.

Les variables numérotées (`$1`, `$2`, `$3`, etc.) et les variables spéciales ci-dessus (`$+`, `&`, `$'`, `$'` et `$^N`) ont toutes une portée dynamique allant jusqu'à la fin du bloc englobant ou jusqu'à la prochaine reconnaissance réussie selon ce qui arrive en premier. (Voir Instructions composées in *perlsyn*.)

**Note** : en Perl, l'échec d'une tentative de reconnaissance ne réinitialise pas ces variables. Cela facilite l'écriture de codes qui testent une succession de cas de plus en plus spécifiques et qui mémorisent la meilleure reconnaissance.

**Attention** : à partir du moment où perl voit que vous avez besoin de l'une des variables `&`, `$'` ou `$'` quelque part dans votre programme, il les calculera à chaque reconnaissance de motif et ce pour tous les motifs. Cela peut ralentir considérablement votre programme. Le même mécanisme est utilisé lors de l'utilisation de `$1`, `$2`, etc.. Ce ralentissement a donc lieu aussi pour les motifs mémorisant des sous-motifs. (Pour éviter ce ralentissement tout en conservant la possibilité d'utiliser des regroupements, utilisez l'extension `(?:...)` à la place.) Mais si vous n'utilisez pas `&`, etc. dans vos scripts alors vos motifs *sans* mémorisation ne seront pas pénalisés. Donc, évitez `&`, `$'` et `$'` si vous le pouvez. Dans le cas contraire (et certains algorithmes en ont réellement besoin), si vous les utilisez une fois, utilisez-les partout car vous en supportez déjà le coût. Depuis la version 5.005, `&` n'est plus aussi coûteux que les deux autres.

Les méta-caractères précédés d'un caractère barre oblique inversée en Perl sont alphanumériques tels `\b`, `\w`, `\n`. À l'inverse d'autres langages d'expressions rationnelles, il n'y a pas de symbole précédé d'un caractère barre oblique inversée qui ne soit pas alphanumérique. Donc tout ce qui ressemble à `\\`, `\(`, `\)`, `\<`, `\>`, `\{`, ou `\}` est toujours interprété littéralement et non comme un méta-caractère. Ceci est utilisé pour désactiver (ou «quoter») les éventuels méta-caractères présents dans une chaîne que vous voulez utiliser comme motif. Tout simplement, il vous suffit de précéder tous les caractères non-"mot" par un caractère barre oblique inversée :

```
$pattern =~ s/(\\W)/\\$1/g;
```

(Si `use local` est actif alors le résultat dépend de votre locale courant.) Aujourd'hui, il est encore plus simple d'utiliser soit la fonction `quotemeta()` soit la séquence `\Q` pour désactiver les éventuels méta-caractères :

```
/\$unquoted\Q$quoted\E$unquoted/
```

Sachez que si vous placez une barre oblique inversée directement (pas celles qui sont dans des variables interpolées) entre `\Q` et `\E`, la double interpolation peut vous amener des résultats très étonnants. Si vous avez *besoin* de constructions de ce genre, consultez Les détails sordides de l'interprétation des chaînes in *perlop*

### 38.1.2 Motifs étendus

Perl intègre une extension de la syntaxe des expressions rationnelles afin d'ajouter les fonctionnalités inexistantes dans les outils standard tels que **awk** et **lex**. La syntaxe est une paire de parenthèses avec un point d'interrogation comme premier caractère entre les parenthèses. Le caractère qui suit le point d'interrogation précise l'extension choisie.

La stabilité de ces extensions est très variable. Certaines font partie du langage depuis de longues années. D'autres sont encore expérimentales et peuvent changer sans préavis ou même être retirées complètement. Vérifier la documentation de chacune de ces extensions pour connaître leur état.

Le point d'interrogation a été choisi pour les extensions ainsi que pour les modificateurs non gourmands parce que 1) les points d'interrogation sont rares dans les vieilles expressions rationnelles et 2) pour qu'à chaque fois que vous en voyez un, vous vous arrêtez pour vous "interroger" sur son comportement. C'est psychologique...

#### (?#texte)

Un commentaire. Le texte est ignoré. Si le modificateur `/x` est utilisé pour autoriser la mise en forme avec des blancs, un simple `#` devrait suffire. Notez que Perl termine le commentaire dès qu'il rencontre `)`. Il n'y a donc aucun moyen de mettre le caractère `)` dans ce commentaire.

**(?imsx-imsx)**

Un ou plusieurs modificateurs de motifs à activer (ou à désactiver s'ils sont précédés par -) jusqu'à la fin du motif ou du sous-motif courant. C'est très pratique pour les motifs dynamiques tels ceux lus depuis un fichier de configuration, ceux provenant d'un argument ou ceux qui sont spécifiés dans une table quelque part. Certains devront être sensibles à la casse, d'autres non. Dans le cas où un motif ne doit pas être sensible à la casse, il est possible d'inclure (?i) au début du motif. Par exemple :

```
$pattern = "foobar";
if (/$pattern/i) { }

plus flexible :

$pattern = "(?i)foobar";
if (/$pattern/) { }
```

Ces modificateurs sont locaux au groupe englobant (si il existe). Par exemple :

```
((?i) blah) \s+ \1
```

reconnaîtra un mot `blah` répété (*y compris sa casse !*) (en supposant le modificateur `x` et aucun modificateur `i` à l'extérieur du groupe).

**(?:motif)****(?imsx-imsx:motif)**

C'est pour regrouper et non pas mémoriser. Cela permet de regrouper des sous-expressions comme "(") mais sans permettre les références arrière. Donc :

```
@fields = split(/\b(?:a|b|c)\b/)
```

est similaire à

```
@fields = split(/\b(a|b|c)\b/)
```

mais ne produit pas de mémorisations supplémentaires. C'est moins couteux de ne pas mémoriser des caractères si vous n'en avez pas besoin.

Les lettres entre ? et : agissent comme des modificateurs. Voir (?imsx-imsx). Par exemple :

```
/(?s-i:more.*than).*million/i
```

est équivalent à l'expression plus verbeuse

```
/(?:(?s-i)more.*than).*million/i
```

**(?=motif)**

Une assertion de longueur nulle pour tester la présence de quelque chose en avant. Par exemple, /\w+(?=\t)/ reconnaît un mot suivi d'une tabulation sans inclure cette tabulation dans \$&.

**(?!motif)**

Une assertion de longueur nulle pour tester l'absence de quelque chose en avant. Par exemple, /foo(?!bar)/ reconnaît toutes les occurrences de "foo" qui ne sont pas suivies de "bar". Notez bien que regarder en avant n'est pas la même chose que regarder en arrière (!). Vous ne pouvez pas utiliser cette assertion pour regarder en arrière.

Si vous cherchez un "bar" qui ne soit pas précédé par "foo", /(?!foo)bar/ ne vous donnera pas ce que vous voulez. C'est parce que (?!foo) exige seulement que ce qui suit ne soit pas "foo" – et ça ne l'est pas puisque c'est "bar", donc "foobar" sera accepté. Vous devez alors utiliser quelque chose comme /(?!foo)...bar/. Nous disons "comme" car il se peut que "bar" ne soit pas précédé par trois caractères. Vous pouvez alors utiliser /(?:?!foo)...|^.{0,2})bar/. Parfois, il est quand même plus simple de dire :

```
if (/bar/ && $' !~ /foo$/)
```

Pour regarder en arrière, voir plus loin.

**(?<=motif)**

Une assertion de longueur nulle pour tester la présence de quelque chose en arrière. Par exemple, /(?<=\t)\w+/ reconnaît un mot qui suit une tabulation sans inclure cette tabulation dans \$&. Cela ne fonctionne qu'avec un motif de longueur fixe.

**(?<!motif)**

Une assertion de longueur nulle pour tester l'absence de quelque chose en arrière. Par exemple, /(?!bar)foo/ reconnaît toutes les occurrences de "foo" qui ne suivent pas "bar". Cela ne fonctionne qu'avec un motif de longueur fixe.

**(?{ code })**

**ATTENTION** : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Une assertion de longueur nulle permettant l'évaluation de code Perl. Elle est reconnue dans tous les cas et le code n'est pas soumis à interpolation. Actuellement les règles pour déterminer où le code se termine sont quelque peu compliquées.

Cette fonctionnalité, utilisée conjointement avec la variable spéciale  $\$^N$ , permet de mémoriser dans des variables les résultats des sous-groupes reconnues sans se préoccuper du nombre de parenthèses imbriquées. Par exemple :

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
```

À l'intérieur du bloc `(?{ .. })`,  $\$_$  fait référence à la chaîne sur laquelle s'applique l'expression rationnelle. Vous pouvez aussi utiliser `pos()` pour connaître la position actuelle dans cette chaîne.

Le code a une portée correcte dans le sens où, si il y a un retour arrière sur l'assertion (voir Retour arrière (§38.1.3)), tous les changements introduits après la localisation sont annulés. Donc :

```
$_ = 'a' x 8;
m<
 (?{ $cnt = 0 }) # Initialisation de $cnt.
 (
 a
 (?{
 local $cnt = $cnt + 1; # Mise a jour de $cnt,
 # (en tenant compte du retour arriere)
 })
)*
 aaaa
 (?{ $res = $cnt }) # En cas de succes, recopie vers une
 # variable non local-isee.
>x;
```

produit `$res = 4`. Remarquez qu'après la reconnaissance, `$cnt` revient à la valeur 0 affectée globalement puisque nous ne sommes plus dans la portée du bloc où l'appel à `local` est effectué.

Cette assertion peut être utilisée comme sélecteur : `(?(condition)motif-oui|motif-non)`. Si elle n'est pas utilisée comme cela, le résultat de l'évaluation du code est affecté à la variable spéciale  $\$^R$ . L'affectation est immédiate donc  $\$^R$  peut être utilisée dans une autre assertion de type `(?{ code })` à l'intérieur de la même expression rationnelle.

L'affectation à  $\$^R$  est correctement localisée, par conséquent l'ancienne valeur de  $\$^R$  est restaurée en cas de retour arrière (voir Retour arrière (§38.1.3)).

Pour des raisons de sécurité, cette construction n'est pas autorisée si l'expression rationnelle nécessite une interpolation de variables lors de l'exécution sauf si vous utilisez la directive `use re 'eval'` (voir *re*) ou si la variable contient le résultat de l'opérateur `qr()` (voir `qr/STRING/imosx` en *perlop*).

Cette restriction est due à l'usage très courant et remarquablement pratique de motifs déterminés lors de l'exécution. Par exemple :

```
$re = <>;
chomp $re;
$string =~ /$re/;
```

Avant que Perl sache comment exécuter du code interpolé à l'intérieur d'un motif, cette opération était complètement sûre d'un point de vue sécurité bien qu'elle puisse générer une exception si le motif n'est pas légal. Par contre, si vous activez la directive `use re 'eval'`, cette construction n'est plus du tout sûre. Vous ne devriez donc le faire que lorsque vous utilisez la vérification des données (via `taint` et l'option `-T`). Mieux encore, vous pouvez utiliser une évaluation contrainte dans un compartiment Safe. Voir *perlsec* pour les détails à propos de ces mécanismes.

**(??{ code })**

**ATTENTION** : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis. Une version simplifiée de la syntaxe pourrait être introduite pour les cas les plus courants.

C'est un sous-motif d'expression rationnelle à "évaluation retardée". Le code est évalué lors de l'exécution au moment où le sous-motif est reconnu. Le résultat de l'évaluation est considéré comme une expression rationnelle dont on doit tenter la reconnaissance comme si elle remplaçait la construction.

Le code n'est pas interpolé. Comme précédemment, les règles pour déterminer l'endroit où se termine le code sont un peu compliquées.

La motif suivant reconnaît des groupes parenthésés :

```
$re = qr{
 \C
 (?
 (:
 (?> [^()]+) # Non-parens without backtracking
 |
 (??{ $re }) # Group with matching parens
)*
 \)
}x;
```

### (?>motif)

**ATTENTION** : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Une sous-expression "indépendante". Reconnaît **uniquement** la sous-chaîne qui aurait été reconnue si la sous-expression avait été seule et ancrée au même endroit. C'est pratique, par exemple pour optimiser certaines constructions qui risqueraient sinon d'être "éternelles", car cette construction n'effectue aucun retour arrière (voir Retour arrière (§38.1.3)). C'est aussi pratique lorsqu'on souhaite le comportement "prend tout ce que tu peux sans jamais redonner quoique ce soit".

Par exemple : `^(?>a*)ab` ne pourra jamais être reconnu puisque `(?>a*)` (ancré au début de la chaîne) reconnaîtra *tous* les caractères a du début de la chaîne en ne laissant aucun a pour reconnaître `ab`. A contrario, `a*ab` reconnaîtra la même chose que `a+b` puisque la reconnaissance du sous-groupe `a*` est influencé par le groupe suivant `ab` (voir Retour arrière (§38.1.3)). En particulier, `a*` dans `a*ab` reconnaît moins de caractères que `a*` seul puisque cela permet à la suite d'être reconnue.

Un effet similaire à `(?>motif)` peut être obtenu en écrivant `(?=(motif))\1`. La première partie de l'expression reconnaît la même sous-chaîne qu'une expression `a+` seule et ensuite le `\1` mange la chaîne reconnue et transforme donc l'assertion de longueur nulle en une expression analogue à `(?>...)`. (La seule différence entre ces deux expressions est que la dernière utilise un groupe mémorisé et décale donc le numéro des références arrières dans le reste de l'expression rationnelle.)

Supposons le motif suivant :

```
m{ \C
 (
 [^()]+ # x+
 |
 \C [^()]* \)
)+
}
```

L'exemple précédent reconnaît de manière efficace un groupe non-vide qui contient au plus deux niveaux de parenthèses. Par contre, si un tel groupe n'existe pas, cela prendra un temps potentiellement infini sur une longue chaîne car il existe énormément de manières différentes de découper une chaîne en sous-chaînes. C'est ce que fait `(.+)+` qui est similaire à l'un des sous-motifs du motif précédent. Rendez-vous compte que l'expression précédente détecte la non reconnaissance sur `((()aaaaaaaaaaaaaaaaaaaaa` en quelques secondes mais que chaque lettre supplémentaire multiplie ce temps par deux. Cette augmentation exponentielle du temps d'exécution peut faire croire que votre programme est planté. Par contre, le même motif très légèrement modifié :

```
m{ \C
 (
 (?> [^()]+) # remplacement de x+ par (?> x+)
 |
 \C [^()]* \)
)+
}
```

en utilisant `(?>...)`, reconnaît exactement la même chose (un bon exercice serait de le vérifier vous-même) mais se termine en 4 fois moins de temps sur une chaîne similaire contenant 1000000 a. Attention, ce motif produit

actuellement un message d'avertissement avec `-w` disant "matches null string many times in regexp" ("reconnait la chaîne de longueur nulle très souvent dans une regexp").

Dans des motifs simples comme `(?> [^()]+ )`, un effet comparable peut être observé en utilisant le test d'absence en avant comme dans `[^()]+ (?! [^()])`. Ce n'est que 4 fois plus lent sur une chaîne contenant 1000000 a.

La comportement "prend tout ce que tu peux sans jamais redonner quoique ce soit" est utile dans de nombreuses situations où une première analyse laisse à penser qu'un simple motif `()*` suffit. Supposez que vous voulez analyser un texte avec des commentaires délimités par `#` suivi d'éventuels espaces (horizontaux). Contrairement aux apparences, `#[ \t]*` n'est *pas* le sous-motif correct pour reconnaître le délimiteur de commentaires parce qu'il peut laisser échapper quelques espaces si la suite du motif en a besoin pour être reconnue. Le réponse correcte est l'une de celles qui suivent :

```
(?>#[\t]*)
#[\t]*(?![\t])
```

Par exemple, pour obtenir tous les commentaires non vides dans `$1`, il vous faut utiliser l'une de ces constructions :

```
/ (?> \# [\t]*) (.+) /x;
/ \# [\t]* ([^ \t] .*) /x;
```

La meilleure des deux est celle qui correspondrait le mieux à la description des commentaires faite par les spécifications.

**(?(condition)motif-oui|motif-non)**

**(?(condition)motif-oui)**

**ATTENTION** : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Expression conditionnelle. `(condition)` est soit un entier entre parenthèses (qui est vrai si le sous-motif mémorisé correspondant reconnaît quelque chose), soit une assertion de longueur nulle (test en arrière, en avant ou évaluation).

Par exemple :

```
m{ (\ () ?
 [^()]+
 (? (1) \)
 }x
```

reconnait une suite de caractères tous différents des parenthèses éventuellement entourée d'une paire de parenthèses.

### 38.1.3 Retour arrière

**NOTE** : cette section présente une abstraction approximative du comportement des expressions rationnelles. Pour une vue plus rigoureuse (et compliquée) des règles utilisées pour choisir entre plusieurs reconnaissances possibles, voir *Combinaison d'expressions rationnelles*.

Une particularité fondamentale de la reconnaissance d'expressions rationnelles est liée à la notion de *retour arrière* qui est actuellement utilisée (si nécessaire) par tous les quantificateurs d'expressions rationnelles. À savoir `*`, `*?`, `+`, `++?`, `{n,m}` et `{n,m}?`. Les retours arrière sont parfois optimisés en interne mais les principes généraux exposés ici restent valides.

Pour qu'une expression rationnelle soit reconnue, *toute* l'expression doit être reconnue, pas seulement une partie. Donc si le début de l'expression rationnelle est reconnue mais de telle sorte qu'elle empêche la reconnaissance de la suite du motif, le moteur de reconnaissance revient en arrière pour calculer autrement le début – d'où le nom retour arrière.

Voici un exemple de retour arrière. Supposons que vous voulez trouver le mot qui suit "foo" dans la chaîne "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if (/\b(foo)\s+(\w+)/i) {
 print "$2 suit $1.\n";
}
```

Lors de la reconnaissance, la première partie de l'expression rationnelle `(\b(foo))` trouve un point d'ancrage dès le début de la chaîne et stocke "Foo" dans `$1`. Puis le moteur de reconnaissance s'aperçoit qu'il n'y pas de caractère d'espacement après le "Foo" qu'il a stocké dans `$1` et, réalisant son erreur, il recommence alors un caractère plus loin que cette première tentative. Cette fois, il va jusqu'à la prochaine occurrence de "foo", l'ensemble de l'expression rationnelle est reconnue et vous obtenez comme prévu "table suit foo."

Parfois la reconnaissance minimale peut aider. Imaginons que vous souhaitez reconnaître tout ce qu'il y a entre "foo" et "bar". Tout d'abord, vous écrivez quelque chose comme :

```

$_ = "The food is under the bar in the barn.";
if (/foo(.*?)bar/) {
 print "got <$1>\n";
}

```

qui, de manière inattendue, produit :

```
got <d is under the bar in the >
```

C'est parce que `.*` est gourmand. Vous obtenez donc tout ce qu'il y a entre le *premier* "foo" et le *dernier* "bar". Dans ce cas, la reconnaissance minimale est efficace pour vous garantir de reconnaître tout ce qu'il y a entre un "foo" et le premier "bar" qui suit.

```

if (/foo(.*?)bar/) { print "got <$1>\n" }
got <d is under the >

```

Voici un autre exemple. Supposons que vous voulez reconnaître un nombre à la fin d'une chaîne tout en mémorisant ce qui précède. Vous écrivez donc :

```

$_ = "I have 2 numbers: 53147";
if (/(.*)(\d*)/) { # Rate!
 print "Beginning is <$1>, number is <$2>.\n";
}

```

Cela ne marche pas parce que `.*` est gourmand et engloutit toute la chaîne. Puisque `\d*` peut reconnaître la chaîne vide, toute l'expression rationnelle est reconnue.

```
Beginning is <I have 2 numbers: 53147>, number is <>.
```

Voici quelques variantes dont la plupart ne marche pas :

```

$_ = "I have 2 numbers: 53147";
@pats = qw{
 (.*)(\d*)
 (.*)(\d+)
 (.*?)(\d*)
 (.*?)(\d+)
 (.*)(\d+)$
 (.*?)(\d+)$
 (.*)\b(\d+)$
 (.*\D)(\d+)$
};

for $pat (@pats) {
 printf "%-12s ", $pat;
 if (/$pat/) {
 print "<$1> <$2>\n";
 } else {
 print "FAIL\n";
 }
}

```

qui affichera :

```

(.*)(\d*) <I have 2 numbers: 53147> <>
(.*)(\d+) <I have 2 numbers: 5314> <7>
(.*?)(\d*) <> <>
(.*?)(\d+) <I have > <2>
(.*)(\d+)$ <I have 2 numbers: 5314> <7>
(.*?)(\d+)$ <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>

```

Comme vous pouvez le constater, c'est un peu délicat. Il faut comprendre qu'une expression rationnelle n'est qu'un ensemble d'assertions donnant une définition du succès. Il peut y avoir aucun, un ou de nombreux moyens de répondre à cette définition lorsqu'on l'applique à une chaîne particulière. Et lorsqu'il y a plusieurs moyens, vous devez comprendre le retour arrière pour savoir quelle variété de succès vous obtiendrez.

Avec l'utilisation des assertions de tests d'absence ou de présence, cela peut devenir carrément épineux. Supposons que vous souhaitez retrouver une suite de caractères non numériques suivie par "123". Vous pouvez essayer d'écrire quelque chose comme :

```
$_ = "ABC123";
if (/^D*(?!123)/) { # Rate!
 print "Heu, pas de 123 dans $_\n";
}
```

Mais ça ne fonctionne pas... tout du moins pas comme vous l'espérez. Ça affiche qu'il n'y a pas de 123 à la fin de la chaîne. Voici une présentation claire de ce qui est reconnu par ces motifs contrairement à ce qu'on pourrait attendre :

```
$x = 'ABC123' ;
$y = 'ABC445' ;

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/ ;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/ ;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

Affiche :

```
2: got ABC
3: got AB
4: got ABC
```

On aurait pu croire que le test 3 échouerait puisqu'il semble être une généralisation du test 1. La différence importante entre les deux est que le test 3 contient un quantificateur ( $\backslash D^*$ ) et peut donc effectuer des retours arrière alors que le test 1 ne le peut pas. En fait, ce que demande le test 3 c'est "en partant du début de \$x, peut-on trouver quelque chose qui n'est pas 123 précédé par 0 ou plusieurs caractères autres que des chiffres?". Si on laisse  $\backslash D^*$  reconnaître "ABC", cela entraîne l'échec du motif complet.

Le moteur de reconnaissance met tout d'abord en correspondance  $\backslash D^*$  avec "ABC". Puis il essaie de mettre en correspondance  $(?!123)$  avec "123", ce qui évidemment échoue. Mais puisque un quantificateur ( $\backslash D^*$ ) est utilisé dans l'expression rationnelle, le moteur de reconnaissance peut faire des retours arrière pour tenter une reconnaissance différente afin de reconnaître l'ensemble de l'expression rationnelle.

Le motif veut *vraiment* être reconnu, alors il utilise le mécanisme de retour arrière et limite cette fois l'expansion de  $\backslash D^*$  juste à "AB". Maintenant, il y a réellement quelque chose qui suit "AB" et qui n'est pas "123" : c'est "C123". C'est suffisant pour réussir.

On peut faire la même chose en mixant l'utilisation des assertions de présence et d'absence. Nous dirons alors que la première partie doit être suivie par un chiffre mais doit aussi être suivie par quelque chose qui n'est pas "123". Souvenez-vous que les tests en avant sont des assertions de longueur nulle – ils ne font que regarder mais ne consomment pas de caractères de la chaîne lors de leur reconnaissance. Donc, réécrit comme suit, cela produira le résultat attendu. C'est à dire que le test 5 échouera et le 6 marchera :

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;

6: got ABC
```

En d'autres termes, cela signifie que les deux assertions de longueur nulle consécutives doivent être vraies toutes les deux ensembles, exactement comme quand vous utilisez des assertions prédéfinies :  $/^{\$}/$  est reconnue uniquement si vous êtes simultanément au début ET à la fin de la ligne. La réalité sous-jacente est que la juxtaposition de deux expressions rationnelles signifie toujours un ET sauf si vous écrivez explicitement un OU en utilisant la barre verticale.  $/ab/$  signifie reconnaître "a" ET (puis) reconnaître "b", même si les tentatives de reconnaissance n'ont pas lieu à la même position puisque "a" n'est pas une assertion de longueur nulle (mais de longueur un).

**Avertissement** : certaines expressions rationnelles compliquées peuvent prendre un temps exponentiel pour leur résolution à cause du grand nombre de retours arrière effectués pour essayer d'être reconnue. Par exemple, sans les optimisations internes du moteur d'expressions rationnelles, l'expression suivante calculerait très longtemps :

```
'aaaaaaaaaaa' =~ /((a{0,5}){0,5})*[c]/
```

et si vous utilisiez des `*` au lieu de limiter entre 0 et 5 reconnaissances, elle pourrait alors prendre littéralement un temps infini – ou plutôt jusqu'au dépassement de la capacité de la pile. De plus, ces optimisations ne sont pas toujours applicables. Par exemple, en remplaçant `{0,5}` par `*` dans le groupe externe, plus aucune optimisation n'a lieu et le calcul devient extrêmement long.

Un outil puissant pour optimiser ce genre de choses est ce qu'on appelle les "groupes indépendants" qui n'effectuent pas de retour arrière (voir `(?motif)>`). Remarquez aussi que les assertions de longueur nulle n'effectuent pas de retour arrière puisqu'elles sont considérées dans un contexte "logique" : seul importe qu'elles soient reconnaissables ou non. Pour un exemple de l'influence éventuelle sur la reconnaissance voir `(?motif)>`.

### 38.1.4 Version 8 des expressions rationnelles

Si vous n'êtes pas familier avec la version 8 des expressions rationnelles, vous trouverez ici les règles de reconnaissance de motifs qui n'ont pas été décrites plus haut.

Un caractère se reconnaît lui-même sauf si c'est un *méta-caractère* avec un sens spécial décrit ici ou précédemment. Pour interpréter un méta-caractère littéralement, il faut le préfixer par un `"\"` (c.-à-d., `"\"` reconnaît un `"` au lieu de n'importe quel caractère, `"\"` reconnaît `"`). Une suite de caractères reconnaît cette suite de caractères dans la chaîne à analyser. Le motif `blurfl` reconnaîtra donc `"blurfl"` dans la chaîne à analyser.

Vous pouvez spécifier une classe de caractères en entourant une liste de caractères entre `[]`. Cette classe reconnaîtra l'un des caractères de la liste. Si le premier caractère après le `"["` est `"^"`, la classe reconnaîtra un caractère qui n'est pas dans la liste. À l'intérieur d'une liste, la caractère `"-"` sert à décrire un intervalle. Par exemple `a-z` représente tous les caractères entre `"a"` et `"z"` inclus. Si vous voulez inclure dans la classe le caractère `"-"` ou `"]"` lui-même, mettez le au début de la classe (après un éventuel `"^"`) ou préfixez le par un `"\"`. `"-"` est aussi interprété littéralement si il est placé en fin de classe, juste avant le `"]"`. (Les exemples suivants décrivent tous la même classe de trois caractères: `[-az]`, `[az-]`, `[a\z]`. Ils sont tous différents de `[a-z]` qui décrit une classe contenant 26 caractères.) Notez aussi que si vous essayez d'utiliser `\w`, `\W`, `\s`, `\S`, `\d` ou `\D` comme borne d'un intervalle, ce ne sera pas un intervalle et le `"-"` sera interprété littéralement.

Remarquez aussi que le concept d'intervalle de caractères n'est pas vraiment portable entre différents codages – et même dans un même codage, cela peut produire un résultat que vous n'attendez pas. Un bon principe de base est de n'utiliser que des intervalles qui commencent et se terminent dans le même alphabet (`[a-e]`, `[A-E]`) ou dans les chiffres (`[0-9]`). Tout le reste n'est pas sûr. Dans le doute, énumérez l'ensemble des caractères explicitement.

Certains caractères peuvent être spécifiés avec la syntaxe des méta-caractères comme en C : `"\n"` reconnaît le caractère nouvelle ligne, `"\t"` reconnaît une tabulation, `"\r"` reconnaît un retour chariot, `"\f"` reconnaît un changement de page, etc. Plus généralement, `\nmn`, où `nmn` est une suite de chiffres octaux, reconnaît le caractère dont le code ASCII est `nmn`. De même, `\xnn`, où `nn` est une suite de chiffres hexadécimaux, reconnaît le caractère dont le code ASCII est `nn`. L'expression `\cx` reconnaît le caractère ASCII control-`x`. Pour terminer, le méta-caractère `"."` reconnaît n'importe quel caractère sauf `"\n"` (à moins d'utiliser `/s`).

Vous pouvez décrire une série de choix dans un motif en les séparant par des `"|"`. Donc `fee|fie|foe` reconnaîtra n'importe quel `"fee"`, `"fie"` ou `"foe"` dans la chaîne à analyser (comme le ferait `f(e|i|o)e`). Le premier choix inclut tout ce qui suit le précédent délimiteur de motif (`"("`, `"["` ou le début du motif) jusqu'au premier `"|"` et le dernier choix inclut tout ce qui suit le dernier `"|"` jusqu'au prochain délimiteur de motif. C'est pour cette raison qu'il est courant d'entourer les choix entre parenthèses pour minimiser les risques de mauvaises interprétation de début et de fin.

Les différents choix sont essayés de la gauche vers la droite et le premier choix qui autorise la reconnaissance de l'ensemble du motif est retenu. Ce qui signifie que les choix ne sont pas obligatoirement "gourmand". Par exemple: en appliquant `foo|foot` à `"barefoot"`, seule la partie `"foo"` est reconnue puisque c'est le premier choix essayé et qu'il permet la reconnaissance du motif complet. (Cela peu sembler anodin mais ne l'est pas lorsque vous mémorisez du texte grâce aux parenthèses.)

Souvenez-vous aussi que `"|"` est interprété littéralement lorsqu'il est présent dans une classe de caractères. Donc si vous écrivez `[fee|fie|foe]`, vous recherchez en fait `[fei|o]`.

À l'intérieur d'un motif, vous pouvez mémoriser ce que reconnaît un sous-motif en l'entourant de parenthèses. Plus loin dans le motif, vous pouvez faire référence au `n`-ième sous-motif mémorisé en utilisant le méta-caractère `\n`. Les sous-motifs sont numérotés de droite à gauche dans l'ordre de leurs parenthèses ouvrantes. Une référence reconnaît exactement la chaîne actuellement reconnue par le sous-motif correspondant et non pas n'importe quelle chaîne qu'il aurait pu reconnaître. Par conséquent, `(0|0x)\d*\s1\d*` reconnaîtra `"0x1234 0x4321"` mais pas `"0x1234 01234"` car, même si `(0|0x)` peut reconnaître le 0 dans le second nombre, ce sous-motif reconnaît dans ce cas `"0x"`.



### 38.1.5 AVERTISSEMENT concernant \1 et \$ 1

Quelques personnes ont l'habitude d'écrire des choses comme :

```
$pattern =~ s/(\W)/\\1/g;
```

Dans la partie droite d'une substitution, ce sont des vieilleries pour ne pas choquer les fans de sed mais ce sont de mauvaises habitudes. Parce que du point de vue Perl, la partie droite d'un s/// est considérée comme une chaîne entre guillemets. \1 dans une chaîne entre guillemets signifie normalement control-A. Le sens Unix habituel de \1 n'est repris que dans s///. Par contre, si vous prenez cette mauvaise habitude, vous serez très perturbé si vous ajoutez le modificateur /e :

```
s/(\d+)/ \1 + 1 /eg; # provoque un "warning" avec -w
```

ou si vous essayez :

```
s/(\d+)/\1000/;
```

Vous ne pouvez lever l'ambiguïté en écrivant \{1}000 alors que c'est possible grâce à \${1}000. En fait, il ne faut pas confondre l'opération d'interpolation et l'opération de reconnaissance d'une référence. Ce sont deux choses bien différents dans la partie *gauche* de s///.

### 38.1.6 Répétition de motifs de reconnaissance de longueur nulle

AVERTISSEMENT: ce qui suit est difficile. Cette section nécessiterait une réécriture.

Les expressions rationnelles fournissent un langage de programmation concis et puissant. Comme avec la plupart des autres outils puissants, ce pouvoir peut faire des ravages.

Un "abus de pouvoir" fréquent découle de la possibilité de créer des boucles sans fin avec des expressions rationnelles aussi innocentes que :

```
'foo' =~ m{ (o?)* }x;
```

o? peut être reconnu au début de 'foo' et, puisque la position dans la chaîne n'est pas modifiée par la reconnaissance, o? sera reconnu indéfiniment à cause du modificateur \*. L'utilisation du modificateur //g est un autre moyen courant d'obtenir de telle cycle :

```
@matches = ('foo' =~ m{ o? }xg);
```

ou

```
print "match: <$$>\n" while 'foo' =~ m{ o? }xg;
```

ou encore dans la boucle implicite de split().

En revanche, on sait depuis longtemps que de nombreuses tâches de programmation peuvent vraiment se simplifier grâce à la reconnaissance répétée de sous-expressions éventuellement de longueur nulle. En voici un exemple simple:

```
@chars = split //, $string; # // n'est pas magique pour split
($whitewashed = $string) =~ s/()/ /g; # les parentheses supprime la magie de s// /
```

Donc Perl autorise la construction /()/ qui rompt de force la boucle infinie. Les règles pour les boucles de bas niveau obtenues par les modificateurs gourmands \*+{} sont différentes de celles de haut niveau induites par exemples par /g ou par l'opérateur split().

Les boucles de bas niveau sont interrompues lorsqu'on détecte la répétition d'une expression reconnaissant une sous-chaîne de longueur nulle. Donc

```
m{ (?: LONGUEUR_NON_NULLE| LONGUEUR_NULLE|)* }x;
```

est en fait équivalent à

```
m{ (? : LONGUEUR_NON_NULLE|)*
 |
 (? : LONGUEUR_NULLE|)?
}x;
```

Les boucles de haut niveau se souviennent entre les itérations que la dernière chaîne reconnue était de longueur nulle. Pour briser la boucle infinie, une chaîne reconnue ne peut pas être de longueur nulle si elle l'était la fois précédente. Cette interdiction interfère avec le retour arrière (voir Retour arrière (§38.1.3)) et dans ce cas, la *seconde meilleure* chaîne est choisie si la *meilleure* est de longueur nulle.

Par exemple :

```
$_ = 'bar';
s/\w??/<$&>/g;
```

produit `<><b><<a><<r><>`. À chaque position dans la chaîne, la meilleure chaîne reconnue par le `??` est la chaîne de longueur nulle et la seconde meilleure chaîne est celle reconnue par `\w`. Donc les reconnaissances de longueur nulle alternent avec celles d'un caractère de long.

De manière similaire, pour des `m/( )/g` répétés, la seconde meilleure reconnaissance est un cran plus loin dans la chaîne.

La mémorisation de l'état *précédente reconnaissance de longueur nulle* est liée à chaque chaîne explorée. De plus, elle est réinitialisée à chaque affectation de `pos()`. Les reconnaissances de longueur nulle à la fin de la précédente reconnaissance sont ignorées durant un `split`.

### 38.1.7 Combinaison d'expressions rationnelles

Dans une expression rationnelle, chaque composant élémentaire tel que décrit précédemment (comme `ab` ou `\Z`) peut reconnaître au plus une sous-chaîne à une position donnée de la chaîne examinée. Par contre, dans une expression rationnelle typique, ces composants élémentaires sont combinés entre eux pour faire des expressions rationnelles plus complexes en utilisant les opérateurs de combinaison `ST`, `S|T`, `S*`, etc. (dans ces exemples `S` et `T` sont des expressions rationnelles).

De telles combinaisons peuvent inclure des alternatives, amenant ainsi à un problème de choix : lorsqu'on applique l'expression rationnelle `a|ab` à la chaîne "abc", est-ce "a" ou "ab" qui est reconnu ? Un moyen de décrire le choix effectué passe par le concept de retour arrière (voir Retour arrière (§38.1.3)). Par contre, cette description est de trop bas niveau et vous amène à penser en termes d'une implémentation particulière.

Une autre manière de décrire les choses commence par les notions de "meilleur"/"pire". Toutes les sous-chaînes qui peuvent être reconnues par une expression rationnelle sont triées de la "meilleure" à la "pire" et c'est la meilleure qui est choisie. Cela remplace la question "Qu'est-ce qui est choisi ?" par la question "Quelle la meilleure reconnaissance et quelle est la pire ?".

Pour la plupart des expressions élémentaires, la question ne se pose pas puisqu'au plus une sous-chaîne peut être reconnue à un emplacement donné. Cette section décrit la notion de meilleure/pire pour les opérateurs de combinaison. Dans les descriptions ci-dessous, `S` et `T` désignent des sous-expressions rationnelles.

#### ST

Soit deux correspondances possibles : `AB` et `A'B'`. `A` et `A'` sont deux sous-chaînes qui peuvent être reconnues par `S`. `B` et `B'` sont deux sous-chaînes qui peuvent être reconnues par `T`.

Si `A` est une meilleure correspondance pour `S` que `A'` alors `AB` est une meilleure correspondance pour `ST` que `A'B'`.

Si `A` et `A'` sont similaires alors `AB` est une meilleure correspondance pour `ST` que `A'B'` si `B` est une meilleure correspondance pour `T` que `B'`.

#### S|T

Lorsque `S` peut être reconnu, c'est une meilleure correspondance qui si seul `T` peut correspondre.

L'ordre entre deux correspondances pour `S` est le même que pour `S` seul. Et c'est la même chose pour deux correspondances pour `T`.

#### S{COMPTEUR}

Correspond à `SSS...S` (répété autant que nécessaire).

#### S{min,max}

Correspond à `S{max}|S{max-1}|...|S{min+1}|S{min}`.

#### S?, S\*, S+

Pareil que `S{0, 1}`, `S{0, INFINI}` et `S{1, INFINI}` respectivement.

**S??, S\*?, S+?**

Pareil que  $S\{0, 1\}?$ ,  $S\{0, INFINI\}?$  et  $S\{1, INFINI\}?$  respectivement.

**(?>S)**

Reconnaît la meilleure correspondance pour S et uniquement celle-là.

**(?=S), (?<=S)**

Seul la meilleure correspondance pour S est utilisé. (Cela n'a d'importance que si S est mémorisé et utilisé ensuite via une référence arrière)

**(?!S), (?<!S)**

Pour ces constructions, il n'y a pas d'ordre à décrire puisque seul compte le fait que S est ou non reconnu.

**(??{ EXPR })**

L'ordre est le même que celui de l'expression rationnelle qui est le résultat de EXPR.

**(?(condition)motif-oui|motif-non)**

Souvenez-vous que le choix du motif à reconnaître (entre motif-oui et motif-non) est déjà fait. L'ordre des reconnaissances est le même que celui du motif retenu.

Les principes précédents décrivent l'ordre des reconnaissances à une position donnée. Une règle supplémentaire est nécessaire pour comprendre comment est déterminée la reconnaissance pour une expression rationnelle complète : une reconnaissance à une position donnée est toujours meilleure qu'une reconnaissance à une position plus lointaine.

**38.1.8 Création de moteurs RE spécifiques (customisés)**

La surcharge de constantes (voir *overload*) est un moyen simple d'augmenter les fonctionnalités du moteur RE (le moteur de reconnaissance d'expressions rationnelles).

Imaginons que vous voulez définir une nouvelle séquence  $\backslash Y|$  qui peut être reconnue à la limite entre un espace et un autre caractère. Remarquez que  $(?=\S)(?<!\S)|(?!\\S)(?<=\S)$  reconnaît exactement ce qu'on veut mais nous voulons remplacer cette expression compliquée par  $\backslash Y|$ . Pour ce faire, nous pouvons créer un module customre :

```
package customre;
use overload;

sub import {
 shift;
 die "No argument to customre::import allowed" if @_ ;
 overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'" }

my %rules = ('\\ ' => '\\ ',
 'Y|' => qr/(?=\S)(?<!\S)|(?!\\S)(?<=\S)/);

sub convert {
 my $re = shift;
 $re =~ s{
 \\ (\\ | Y .)
 }
 { $rules{$1} or invalid($re,$1) }sgex;
 return $re;
}
```

Il vous suffit alors d'utiliser `use customre` pour utiliser cette nouvelle séquence dans les expressions rationnelles constantes, c.-à-d. celles qui ne nécessitent pas d'interpolation de variables lors de l'exécution. Telle que documentée dans *overload*, cette conversion ne fonctionne que sur les parties littérales des expressions rationnelles. Dans  $\backslash Y|\$re\backslash Y|$ , la partie variable de cette expression rationnelle doit être convertie explicitement (mais uniquement si la signification spécifique de  $\backslash Y|$  est nécessaire dans \$re) :

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|\$re\Y|/;
```

## 38.2 BUGS

Le niveau de difficulté de ce document varie de "difficile à comprendre" jusqu'à "totalement opaque". La prose divaguante et criblée de jargon est difficile à interpréter en divers endroits.

## 38.3 VOIR AUSSI

Opérateurs d'expression rationnelle in *perlop*.

Les détails sordides de l'interprétation des chaînes in *perlop*.

*perlfaq6*.

pos in *perlfunc*.

*perllocale*.

*perlebcdic*.

*Mastering Regular Expressions* par Jeffrey Friedl, publié chez O'Reilly and Associates.

## 38.4 TRADUCTION

### 38.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 38.4.2 Traducteur

Traduction initiale et mise à jour 5.6.0, 5.6.1 et 5.8.8 par Paul Gaborit <Paul.Gaborit @ enstimac.fr>.

### 38.4.3 Relecture

Régis Julié <Regis.Julie@cetelem.fr>.

# Chapitre 39

## perlref

La référence pour les expressions rationnelles en Perl

### 39.1 DESCRIPTION

Cette page est une référence rapide aux expressions rationnelles de Perl (NdT : on emploie couramment le terme *expression régulière* car le terme anglais est *regular expression* qui s'abrège en *regexp*. Mais ne nous y trompons pas, en français, ce sont bien des *expressions rationnelles* ; nous utiliserons cependant l'abréviation *regexp*). Pour de plus amples informations, voir *perlre* et *perlop*, ainsi que la section VOIR AUSSI (§39.3) de cette page.

#### 39.1.1 OPÉRATEURS

`=~` détermine la variable à laquelle s'applique l'expression rationnelle. En cas d'absence, `$_` est utilisée.

```
$var =~ /foo/;
```

`!~` détermine à quelle variable l'expression rationnelle s'applique, et effectue une négation sur le résultat de la reconnaissance ; retourne faux si la reconnaissance réussit, vrai si elle échoue.

```
$var !~ /foo/;
```

`m/pattern/igmsocx` parcourt une chaîne de caractères à la recherche d'un motif, en appliquant les modificateurs donnés.

- `i` insensible à la casse
- `g` global - toutes les occurrences
- `m` mode multiligne - `^` et `$` reconnaissent les débuts/fin de lignes
- `s` reconnaît comme une seule ligne - `'.'` reconnaît `\n`
- `o` compiler le motif une seule fois
- `x` lisibilité étendue - permet les commentaires
- `c` n'efface pas pos lors de l'échec d'une reconnaissance, lors de l'utilisation de `/g`

Si le motif est une chaîne vide, la dernière *regexp* reconnue avec succès sera utilisée. Il est possible d'utiliser d'autres délimiteurs que `'/'`, aussi bien pour cet opérateur que pour les suivants.

`qr/pattern/imsox` permet de stocker une *regexp* dans une variable ou de la passer en argument. Les modificateurs sont les mêmes que pour l'opérateur `m//` et sont stockés avec la *regexp*.

`s/pattern/replacement/igmsoxe` remplace le motif 'pattern' par le motif 'replacement'. Les modificateurs sont les mêmes que pour l'opérateur `m//`, avec un supplément :

`e` évaluer 'replacement' comme une expression

'e' peut être ajouté plusieurs fois. 'replacement' est interprété comme une chaîne entre double-quotes à moins que le délimiteur soit une simple quote.

`?pattern?` est équivalent à `m/pattern/`, mais le motif n'est reconnu qu'une seule fois. Aucun autre délimiteur ne peut être utilisé. Doit être réinitialisé avec `L<reset|perlfunc/reset>`.

### 39.1.2 SYNTAXE

`\` Permet d'échapper le caractère suivant  
`.` Reconnaît n'importe quel caractère, sauf une fin de ligne (à moins que `/s` soit utilisé)  
`^` Reconnaît le début de la chaîne (ou de la ligne, si `/m` est utilisé)  
`$` Reconnaît la fin de la chaîne (ou de la ligne, si `/m` est utilisé)  
`*` Reconnaît l'élément précédent 0 ou plusieurs fois  
`+` Reconnaît l'élément précédent au moins 1 fois  
`?` Reconnaît l'élément précédent 0 ou 1 fois  
`{...}` Permet de donner un nombre d'occurrences de l'élément précédent  
`[...]` Reconnaît n'importe lequel des caractères contenus dans les crochets  
`(...)` Permet de faire des groupes pour les stocker dans `$1`, `$2`, ...  
`(?:...)` Permet de faire des groupes sans stockage  
`|` Reconnaît soit l'expression précédente, soit l'expression suivant l'opérateur  
`\1`, `\2` ... Le texte du Nième groupe

### 39.1.3 SÉQUENCES D'ÉCHAPPEMENT

Ceux-ci fonctionnent comme des chaînes normales.

`\a` alarme (bip)  
`\e` escape (penser à troff)  
`\f` page suivante  
`\n` nouvelle ligne  
`\r` retour chariot  
`\t` tabulation  
`\037` caractère en octal (penser au PDP-11)  
`\x7f` caractère en hexadécimal  
`\x{263a}` caractère en hexadécimal étendu (Unicode SMILEY)  
`\cx` control-x  
`\N{name}` caractère nommé

`\l` convertit en minuscule le caractère suivant  
`\u` convertit en majuscule le caractère suivant  
`\L` convertit en minuscule jusqu'au prochain `\E`  
`\U` convertit en majuscule jusqu'au prochain `\E`  
`\Q` désactive les méta-caractères de motif jusqu'au prochain `\E`  
`\E` fin de modification de casse

Pour les majuscules, voir Casse de titre.

Celui-ci fonctionne différemment des chaînes normales :

`\b` une assertion, pas le caractère retour en arrière, sauf dans une classe de caractères

### 39.1.4 CLASSES DE CARACTÈRES

[amy]	Reconnaît 'a', 'm' ou 'y'
[f-j]	Le tiret permet la reconnaissance d'un ordre (NdT : les caractères de f à j)
[f-j-]	Un tiret échappé, au début ou à la fin signifie '-'
[^f-j]	Le chapeau indique une négation (reconnaît tous les caractères sauf ceux-ci)

Les séquences suivantes fonctionnent avec ou sans classe de caractères. Les six premiers sont conscients des locales, tous sont conscients d'Unicode. Les classes de caractères équivalentes par défaut sont données. Voir *perllocale* et *perlunicode* pour plus de détails.

<code>\d</code>	Reconnaît un chiffre	<code>[0-9]</code>
<code>\D</code>	Négation de <code>\d</code>	<code>[^0-9]</code>
<code>\w</code>	Reconnaît un caractère de "mot" (alphanumérique plus "_")	<code>[a-zA-Z0-9_]</code>
<code>\W</code>	Négation de <code>\w</code>	<code>[^a-zA-Z0-9_]</code>
<code>\s</code>	Reconnaît un caractère "blanc" (espace, tabulation,...)	<code>[\t\n\r\f]</code>
<code>\S</code>	Négation de <code>\s</code>	<code>[^\t\n\r\f]</code>
<code>\C</code>	Reconnaît un octet (avec Unicode, '.' reconnaît un caractère)	
<code>\pP</code>	Reconnaît la propriété Unicode P	
<code>\p{...}</code>	Reconnaît la propriété Unicode (avec un nom long)	
<code>\PP</code>	Négation de <code>\pP</code>	
<code>\P{...}</code>	Négation de la propriété Unicode (avec un nom long)	
<code>\X</code>	Reconnaît une séquence d'un caractère Unicode étendue, équivalent à <code>(?:\PM\pM*)</code>	

Les classes de caractères POSIX et leurs équivalents Unicode et Perl :

<code>alnum</code>	<code>IsAlnum</code>	Alphanumérique
<code>alpha</code>	<code>IsAlpha</code>	Alphabétique
<code>ascii</code>	<code>IsASCII</code>	Un caractère ASCII
<code>blank</code>	<code>IsSpace</code> <code>[\t]</code>	Un "blanc" horizontal (extension GNU)
<code>cntrl</code>	<code>IsCntrl</code>	Caractère de contrôle
<code>digit</code>	<code>IsDigit</code> <code>\d</code>	Chiffre
<code>graph</code>	<code>IsGraph</code>	Alphanumérique et ponctuation
<code>lower</code>	<code>IsLower</code>	Caractères bas de casse (conscient de locale et d'Unicode)
<code>print</code>	<code>IsPrint</code>	Alphanumérique, point et espace
<code>punct</code>	<code>IsPunct</code>	Ponctuation
<code>space</code>	<code>IsSpace</code> <code>[\s\ck]</code>	Un "blanc"
	<code>IsSpacePerl</code> <code>\s</code>	Un "blanc" au sens de Perl
<code>upper</code>	<code>IsUpper</code>	Caractères "casse de titre" (conscient de locale et d'Unicode)
<code>word</code>	<code>IsWord</code> <code>\w</code>	Alphanumérique et "_" (extension Perl)
<code>xdigit</code>	<code>IsXDigit</code> <code>[0-9A-Fa-f]</code>	Chiffre hexadécimal

Dans une classe de caractères :

POSIX	traditional	Unicode
<code>[:digit:]</code>	<code>\d</code>	<code>\p{IsDigit}</code>
<code>[:^digit:]</code>	<code>\D</code>	<code>\P{IsDigit}</code>

### 39.1.5 ANCRES

Toutes sont des assertions de longueur nulle.

```

^ Reconnait le début de la chaîne (ou de la ligne, si /m est utilisé)
$ Reconnait la fin de la chaîne (ou de la ligne, si /m est utilisé)
 ou avant une nouvelle ligne
\b Reconnait une limite de mot (entre \w et \W)
\B Négation de \b (entre \w et \w ou entre \W et \W)
\A Reconnait un début de chaîne (avec ou sans /m)
\Z Reconnait une fin de chaîne (avant d'éventuels débuts de ligne)
\z Reconnait la "vraie" fin de chaîne
\G Reconnait l'endroit où le précédent m//g s'est arrêté

```

### 39.1.6 QUANTIFICATEURS

Les quantificateurs sont gourmands par défaut – ils reconnaissent le **plus long** motif possible.

Maximum	Minimum	Signification
{n,m}	{n,m}?	Reconnait au moins n fois mais pas plus de m fois
{n,}	{n,}?	Reconnait au moins n fois
{n}	{n}?	Reconnait exactement n fois
*	*?	Reconnait 0 fois ou plus (identique à {0,})
+	+?	Reconnait 1 fois ou plus (identique à {1,})
?	??	Reconnait 0 fois ou 1 (identique à {0,1})

Il n'existe pas de quantificateur {,n} – il est interprété comme une chaîne.

### 39.1.7 CONSTRUCTIONS ÉTENDUES

```

(?#text) Un commentaire
(?imxs-imsx:...) Activer/Désactiver une option (comme les
 modificateurs pour m//)
(?=...) Assertion de longueur nulle pour tester la présence
 de quelque chose en avant
(?!...) Assertion de longueur nulle pour tester l'absence
 de quelque chose en avant
(?<=...) Assertion de longueur nulle pour tester la présence
 de quelque chose en arrière
(?<!...) Assertion de longueur nulle pour tester l'absence
 de quelque chose en arrière
(?>...) Capturer tout ce qui est possible, sans retours arrière
(?{ code }) Code embarqué, la valeur de retour devient $^R
(??{ code }) Regexp dynamique, la valeur de retour est une regexp
?(cond)yes|no (cond) est soit un entier entre parenthèses (qui est
?(cond)yes vrai si le sous-motif mémorisé correspondant reconnaît
 quelque chose), soit une assertion de longueur nulle
 (test en arrière, en avant ou évaluation)

```

### 39.1.8 VARIABLES

```

$_ Variable utilisée par défaut par les opérateurs
*$ Activer la reconnaissance multi-ligne (obsolète, disparaît à
 partir de 5.9.0)

& Totalité de la chaîne reconnue
$' Tout ce qui précède la chaîne reconnue
$' Tout ce qui suit la chaîne reconnue

```



L'utilisation de ces trois derniers opérateurs va ralentir **toutes** les regexps utilisées dans votre programme. Vous pouvez consulter *perlvar* (@LAST\_MATCH\_START) pour trouver des expressions équivalentes ne ralentissant pas le programme. Voir aussi *Devel::SawAmperand*.

```
$1, $2 ... contiennent la Xième expression capturée
$+ La dernière expression reconnue entre parenthèses
$^N Contient la plus récente des captures fermées
$^R Contient le résultat de la dernière expression (?{...})
@- Contient les décalages de début de groupes. $-[0] contient le
 décalage depuis le début de toute la reconnaissance
@+ Contient les décalages de fin de groupes. $+[0] contient le
 décalage de la fin de toute la reconnaissance
```

Les groupes capturés sont numérotés selon l'ordre de leur parenthèse *ouvrante*.

### 39.1.9 FONCTIONS

```
lc Mettre une chaîne en bas de casse
lcfirst Mettre le premier caractère de la chaîne en bas de casse
uc Mettre une chaîne en casse de titre
ucfirst Mettre le premier caractère de la chaîne en casse de titre

pos Renvoie ou fixe la position courante de la reconnaissance
quotemeta Citer des méta-caractères
reset Effacer le statut de ?pattern?
study Analyser une chaîne pour optimiser la reconnaissance

split Utiliser une regexp pour diviser une chaîne en plusieurs
 morceaux
```

Les quatre premières fonctions correspondent aux séquences d'échappement \L, \l, \U et \u. Pour la casse de titre, cf. Casse de titre.

### 39.1.10 TERMINOLOGIE

#### Casse de titre

C'est un concept Unicode qui est généralement équivalent au haut de casse, excepté pour certains caractères comme le "Estset" allemand.

## 39.2 AUTEUR

Iain Truskett.

Ce document peut être distribué dans les mêmes conditions que Perl lui-même.

## 39.3 VOIR AUSSI

- *perlretut* pour un tutoriel sur les expressions rationnelles.
- *perlrequick* pour un tutoriel rapide sur les regexp.
- *perlre* pour plus de détails.
- *perlvar* pour plus de détails sur les variables.
- *perlop* pour plus de détails sur les opérateurs.
- *perlfunc* pour plus de détails sur les fonctions.
- *perlfag6* pour la FAQ sur les expressions rationnelles.
- Le module *re* pour modifier le comportement et aider au débogage.
- Debugging regular expressions in *perldebug*
- *perluniintro*, *perlunicode*, *chardnames* et *locale* pour plus de détails sur les regexp et l'internationalisation.
- *Mastering Regular Expressions* by Jeffrey Friedl (<http://regex.info/>) pour un approfondissement et une documentation plus complète du sujet.

## **39.4 REMERCIEMENTS**

David P.C. Wollmann, Richard Soderberg, Sean M. Burke, Tom Christiansen, Jim Cromie et Jeffrey Goff pour leurs précieux conseils.

## **39.5 TRADUCTION**

### **39.5.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **39.5.2 Traducteur**

Traduction depuis 5.8.8 par Thomas van Oudenhove.

### **39.5.3 Relecture**

Aucune pour l'instant.

# Chapitre 40

## perlref

Références et structures de données imbriquées en Perl

### 40.1 NOTE

Ce document est la documentation complète abordant tous les aspects des références. Pour une introduction n'abordant que les fonctionnalités essentielles et donc plus courte et surtout plus pédagogique, voir *perlref.tut*.

### 40.2 DESCRIPTION

Avant la version 5 de Perl, il était difficile de représenter des structures de données complexes car toutes les références devaient être symboliques (et même dans ce cas, il était difficile de référencer une variable à la place d'une entrée symbolique de tableau). Désormais, non seulement Perl facilite l'utilisation de références symboliques à des variables, mais il vous laisse en plus la possibilité d'avoir des références "dures" à tout morceau de données ou de code. N'importe quel scalaire peut contenir une référence dure. Comme les tableaux et les tables de hachage contiennent des scalaires, vous pouvez désormais construire facilement des tableaux de tableaux, des tableaux de tables de hachage, des tables de hachage de tableau, des tableaux de tables de hachage de fonctions, etc.

Les références dures sont intelligentes : elles conservent la trace du nombre de références pour vous, libérant automatiquement la structure référencée quand son compteur de références atteint zéro. (Le compteur de références pour des valeurs dans des structures de données auto-référencées ou cycliques ne pourra pas atteindre zéro sans un petit coup de pouce. Cf. Ramasse-miettes à deux phases in *perlobj* pour une explication détaillée.) Si cette structure s'avère être un objet, celui-ci est détruit. Cf. *perlobj* pour de plus amples renseignements sur les objets. (Dans un certain sens, tout est objet en Perl, mais d'habitude nous réservons ce mot pour les références à des structures qui ont été officiellement "bénies" dans un paquetage de classes.

Les références symboliques sont des noms de variables ou d'autres objets, tout comme un lien symbolique dans un système de fichiers Unix ne contient à peu de choses près que le nom d'un fichier. La notation `*glob` est un type de référence symbolique. (Les références symboliques sont parfois appelées "références douces" mais évitez de les appeler ainsi ; les références sont déjà suffisamment confuses sans ces synonymes inutiles.)

Au contraire, les références dures ressemblent plus aux liens durs dans un système de fichiers Unix : elles sont utilisées pour accéder à un objet sous-jacent sans se préoccuper de son (autre) nom. Quand le mot "référence" est utilisé sans adjectif, comme dans le paragraphe suivant, il est habituellement question d'une référence dure.

Les références sont faciles à utiliser en Perl. Il n'existe qu'un principe majeur : Perl ne référence et ne déréférence jamais de façon implicite. Quand un scalaire contient une référence, il se comporte toujours comme un simple scalaire. Il ne devient pas magiquement un tableau, une table de hachage ou une routine. Vous devez le lui préciser explicitement, en le déréférençant.

#### 40.2.1 Créer des références

Les références peuvent être créées de plusieurs façons.

1. En utilisant l'opérateur "backslash" [barre oblique inverse, ndt] sur une variable, une routine ou une valeur. (Cela fonctionne plutôt comme l'opérateur & (adresse de), du C.) Notez bien que cela crée typiquement une *AUTRE* référence à la variable, parce qu'il existe déjà une telle référence dans la table des symboles. Mais même si la référence de la table des symboles disparaît, vous aurez toujours la référence que la backslash a retournée. Voici quelques exemples :

```
$scalarref = \ $foo;
$arrayref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;
$globref = *foo;
```

Il est impossible de créer une véritable référence à un descripteur d'E/S (descripteur de fichier ou de répertoire) en utilisant l'opérateur backslash. Le mieux que vous puissiez obtenir est une référence à un typeglob, qui est en fait une entrée complète de la table des symboles. Voir l'explication de la syntaxe `*foo{THING}` ci-dessous. Quoi qu'il en soit, vous pouvez toujours utiliser les typeglobs et les globrefs comme s'il étaient des descripteur d'E/S.

2. Une référence à un tableau anonyme peut être créée en utilisant des crochets :

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

Ici, nous avons créé une référence à un tableau anonyme de trois éléments, dont le dernier est lui-même une référence à un autre tableau anonyme de trois éléments. (La syntaxe multidimensionnelle décrite plus loin peut être utilisée pour y accéder. Par exemple, après le code ci-dessus, `$arrayref->[2][1]` aura la valeur "b".)

Prendre une référence à une liste énumérée n'est pas la même chose que d'utiliser des crochets (c'est plutôt la même chose que créer une liste de références !)

```
@list = (\ $a, \@b, \%c);
@list = (\ $a, @b, %c); # identique !
```

À l'exception de `\(@foo)` qui retourne une liste de références au contenu de `@foo`, et non pas une référence à `@foo` lui-même. Il en est de même pour `%foo` sauf évidemment pour les clés elle-mêmes qui seront simplement recopiées (puisque les clés sont justes des chaînes de caractères et non des scalaires au sens large).

3. Une référence à une table de hachage anonyme peut être créée en utilisant des accolades :

```
$hashref = {
 'Adam' => 'Eve',
 'Clyde' => 'Bonnie',
};
```

Les composants de table de hachage et de tableau comme ceux-ci peuvent être librement mélangés pour produire une structure aussi complexe que vous le souhaitez. La syntaxe multidimensionnelle décrite ci-dessous fonctionne pour ces deux cas. Les valeurs ci-dessus sont littérales mais des variables et expressions fonctionneraient de la même manière, car l'opérateur d'affectation en Perl (même à l'intérieur d'un `local()` ou d'un `my()`) sont des instructions exécutables et non pas des déclarations à la compilation.

Comme les accolades sont utilisées pour bien d'autres choses, y compris les BLOCs, vous pourriez être amené à devoir expliciter les accolades au début d'une instruction en ajoutant un `+` ou un `return` devant, de telle sorte que Perl comprenne que l'accolade ouvrante n'est pas le commencement d'un BLOC. Les économies réalisées et la valeur mnémotechnique des accolades valent bien cet embarras supplémentaire.

Par exemple, si vous désirez une fonction qui crée une nouvelle table de hachage et retourne une référence à celle-ci, vous avez ces possibilités :

```
sub hashem { { @_ } } # silencieusement faux
sub hashem { +{ @_ } } # correct
sub hashem { return { @_ } } # correct
```

D'un autre côté, si vous souhaitez l'autre signification, vous pouvez faire ceci :

```
sub showem { { @_ } } # ambigu (correct pour le moment
 # mais pourrait changer)
sub showem { {; @_ } } # correct
sub showem { { return @_ } } # correct
```

Les `+` et `{;` en début servent à différencier de manière explicite soit une référence à un TABLE DE HACHAGE soit un BLOC.

4. Une référence à une routine anonyme peut être créée en utilisant `sub` sans nom de routine :

```
$coderef = sub { print "Boink!\n" };
```

Notez la présence du point-virgule. À part le fait que le code à l'intérieur n'est pas exécuté immédiatement, un `sub {}` n'est ni plus ni moins qu'une déclaration comme opérateur, tout comme `do{} ou eval{}.` (Peu importe le nombre de fois que vous allez exécuter cette ligne particulière – à moins que vous soyez dans un `eval("...")` – `$coderef` fera toujours référence à la *MÊME* routine anonyme.)

Les routines anonymes fonctionnent comme les fermetures, en respectant les variables `my()`, c'est-à-dire les variables lexicalement visibles dans la portée actuelle. La fermeture est une notion provenant de l'univers Lisp qui indique que, si vous définissez une fonction anonyme dans un contexte lexical particulier, elle essaiera de fonctionner dans ce contexte, même quand elle est appelée en-dehors de ce contexte.

En termes plus humains, c'est une façon amusante de passer des arguments à une routine, aussi bien lorsque vous la définissez que lorsque vous l'appellez. C'est très utile pour mettre au point des petits morceaux de code à exécuter plus tard, comme les callbacks. Vous pouvez même faire de l'orienté objet avec ça bien que Perl fournisse déjà un autre mécanisme pour le faire (voir *perlobj*).

Vous pouvez aussi considérer la fermeture comme une façon d'écrire un modèle de routine sans utiliser `eval()`. Voici un petit exemple de fonctionnement des fermetures :

```
sub newprint {
 my $x = shift;
 return sub { my $y = shift; print "$x, $y !\n"; };
}
$h = newprint("Bonjour");
$g = newprint("Salutations");

Un ange passe...

&$h("monde");
&$g("humains");
```

Ce qui affiche

```
Bonjour, monde !
Salutations, humains !
```

Notez en particulier que `$x` continue à référencer la valeur passée à `newprint()` *bien que* le `"my $x"` semble être hors de la portée au moment où la routine anonyme est exécutée. Voici donc ce qu'est la fermeture.

À propos, ceci ne s'applique qu'aux variables lexicales. Les variables dynamiques continuent de fonctionner comme elle l'ont toujours fait. La fermeture n'est pas une chose dont la plupart des programmeurs Perl ont besoin de s'embarrasser pour commencer.

5. Les références sont souvent retournées par des routines spéciales appelées constructeurs. Les objets Perl sont juste des références à un type particulier d'objet qui s'avère capable de connaître quel paquetage y est associé. Les constructeurs sont juste des routines particulières qui savent comment créer cette association. Ils le font en commençant par une référence ordinaire qui reste telle quelle même si c'est un objet. Les constructeurs sont souvent nommés `new()` et appelés indirectement :

```
$objref = new Doggie (Tail => 'short', Ears => 'long');
```

Mais il n'est pas nécessaire d'avoir :

```
$objref = Doggie->new(Tail => 'short', Ears => 'long');

use Term::Cap;
$terminal = Term::Cap->Tgetent({ OSPEED => 9600 });

use Tk;
$main = MainWindow->new();
$menu = $main->Frame(-relief => "raised",
 -borderwidth => 2)
```

6. Des références de type approprié peuvent venir à exister si vous les déréférenciez dans un contexte qui suppose qu'elles existent. Comme nous n'avons pas encore parlé du déréférencement, nous ne pouvons toujours pas vous montrer d'exemples.
7. Une référence peut être créée en utilisant une syntaxe particulière, sentimentalement connue comme la syntaxe `*foo{THING}`. `*foo{THING}` retourne une référence à l'emplacement `THING` dans `*foo` (qui est l'entrée de la table des symboles contenant tout ce qui est connu en tant que "foo").

```

$scalarref = *foo{SCALAR};
$arrayref = *ARGV{ARRAY};
$hashref = *ENV{HASH};
$coderef = *handler{CODE};
$ioref = *STDIN{IO};
$globref = *foo{GLOB};
$formatref = *foo{FORMAT};

```

Tout ceci s'explique de lui-même, à part `*foo{IO}`. Il retourne le descripteur d'E/S utilisé pour les descripteurs de fichiers (`open` in *perlfunc*), de sockets (`socket` in *perlfunc* et `socketpair` in *perlfunc*) et de répertoires (`opendir` in *perlfunc*). Pour des raisons de compatibilités avec les versions précédentes de Perl, `*foo{FILEHANDLE}` est un synonyme de `*foo{IO}`. Si les avertissements sont actifs, l'utilisation de ce synonyme affichera un message.

`*foo{TURC}` retourne un indéfini si ce TRUC particulier n'a pas été utilisé auparavant, sauf dans le cas des scalaires. `*foo{SCALAR}` retourne une référence à un scalaire anonyme si `$foo` n'a pas encore été utilisé. Ceci pourrait changer dans une prochaine version.

`*foo{IO}` est une autre manière d'accéder au mécanisme `\*HANDLE` indiqué dans `Typeglobs` et `Handles de Fichiers` in *perldata* pour passer des descripteurs de fichiers comme arguments ou comme valeur de retour de routines, ou pour les stocker dans des structures de données plus grandes. L'inconvénient, c'est qu'il ne crée pas de nouveau descripteur de fichier pour vous. L'avantage, c'est qu'il y a moins de risque d'aller au-delà de ce que vous souhaitez qu'avec une affectation de typeglob (il passe tout de même les descripteurs de fichier et de répertoire). Ceci étant, si vous l'affectez à un scalaire au lieu d'un typeglob comme dans l'exemple ci-dessous, vous êtes couvert dans tous les cas.

```

splutter(*STDOUT); # passe tout le glob
splutter(*STDOUT{IO}); # ne passe que le descripteur
 # de fichier et de répertoire

sub splutter {
 my $fh = shift;
 print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN); # passe tout le glob
$rec = get_rec(*STDIN{IO}); # ne passe que le descripteur
 # de fichier et de répertoire

sub get_rec {
 my $fh = shift;
 return scalar <$fh>;
}

```

### 40.2.2 Utiliser des références

C'est tout pour la création de références. Maintenant, vous devez sûrement mourir d'envie de savoir comment utiliser ces références pour en revenir à vos données perdues depuis longtemps. Il existe plusieurs méthodes de base.

1. Où que vous mettiez un identifiant (ou une chaîne d'identifiants) comme partie d'une variable ou d'un nom de routine, vous pouvez remplacer cet identifiant par une simple variable scalaire contenant une référence de type correct :

```

$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";

```

Il est important de comprendre qu'ici, nous ne déréférençons *pas* en particulier `$arrayref[0]` ou `$hashref{"KEY"}`. Le déréférencement de la variable scalaire a lieu *avant* toute recherche de clé. Tout ce qui est plus complexe qu'une simple variable scalaire doit utiliser les méthodes 2 et 3 ci-dessous. un "simple scalaire" inclut toutefois un identifiant qui utilise lui-même la méthode 1 de façon récursive. Le code suivant imprime par conséquent "howdy".

```
$refrefref = \\\"howdy\";
print $$$refrefref;
```

2. Où que vous mettiez un identifiant (ou une chaîne d'identifiants) comme partie d'une variable ou d'un nom de routine, vous pouvez remplacer cet identifiant par un BLOC retournant une référence de type correct. En d'autres mots, les exemples précédents auraient pu être écrits ainsi :

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
$globref->print("output\n"); # ssi IO::Handle est chargé
```

Il est vrai que c'est un peu idiot d'utiliser des accolades dans ce cas-là, mais le BLOC peut contenir n'importe quelle expression, en particulier une expression subscript telle que celle-ci :

```
&{ $dispatch{$index} }(1,2,3); # appel la routine correcte
```

Comme il est possible d'omettre les accolades dans le cas simple de `$$x`, les gens font souvent l'erreur de considérer le déréférencement comme des opérateurs propres et se posent des questions à propos de leur précedence. Mais s'ils en étaient, vous pourriez utiliser des parenthèses à la place des accolades. Ce qui n'est pas le cas. Remarquez la différence ci-dessous. Le cas 0 est un raccourci du cas 1 mais *pas* du cas 2.

```
$$hashref{"KEY"} = "VALUE"; # CAS 0
${$hashref}{"KEY"} = "VALUE"; # CAS 1
${$hashref{"KEY"}} = "VALUE"; # CAS 2
${$hashref->{"KEY"}} = "VALUE"; # CAS 3
```

Le cas 2 est aussi décevant dans le sens que vous accédez à une variable appelée `%hashref`, sans déréférencer par `$hashref` la table de hachage qu'il référence probablement. Ceci correspond au cas 3.

3. Les appels de routines et les recherches d'éléments individuels de tableaux sont tellement courants qu'il devient pénible d'utiliser la méthode 2. En forme de sucre syntaxique, les exemples de la méthode 2 peuvent être écrits ainsi :

```
$arrayref->[0] = "January"; # Élément de tableau
$hashref->{"KEY"} = "VALUE"; # Élément de table de hachage
$coderef->(1,2,3); # Appel d'une routine
```

La partie gauche de la flèche peut être n'importe quelle expression retournant une référence, y compris un déréférencement précédent. Notez que `$array[$x]` n'est *pas* ici la même chose que `$array->[$x]` :

```
$array[$x]->{"foo"}->[0] = "January";
```

C'est un des cas que nous avons mentionnés plus tôt et dans lequel les références peuvent venir à exister dans un contexte lvalue. Avant cette instruction, `$array[$x]` peut avoir été indéfini. Dans ce cas, il est automatiquement défini avec une référence de table de hachage, de telle sorte que nous puissions rechercher `{"foo"}` dedans. De la même manière, `$array[$x]->{"foo"}` sera automatiquement défini avec une référence de tableau, de telle façon que nous puissions rechercher dedans. Ce processus est appelé *autovivification*.

Encore une petite chose ici : la flèche est facultative *entre* les crochets subscripts. Vous pouvez donc abréger le code ci-dessus en :

```
$array[$x>{"foo"}[0] = "January";
```

Ce qui, dans le cas dégénéré de la seule utilisation de tableaux ordinaires, vous donne des tableaux multidimensionnels tout comme en C :

```
$score[$x][$y][$z] += 42;
```

Bon, d'accord, pas complètement comme en C, en fait. Le C ne sait pas comment agrandir ses tableaux à la demande. Perl le sait.

4. Si une référence se révèle être une référence à un objet, il existe alors probablement des méthodes pour accéder aux choses référencées, et vous devriez vous cantonner à ces méthodes à moins que vous ne soyez dans le packaging de classes qui définit justement les méthodes de cet objet. En d'autres termes, soyez sages et ne violez pas l'encapsulation des objets sans d'excellentes raisons. Perl ne renforce pas l'encapsulation. Nous ne sommes pas totalitaires. En revanche, nous attendons un minimum de politesse.

L'utilisation d'un nombre ou d'une chaîne en tant que référence en fait une référence symbolique comme expliqué plus haut. L'utilisation d'une référence en tant que nombre la transforme en un entier représentant son emplacement en mémoire. Le seul usage intéressant est la comparaison numérique de deux références pour savoir si elles se réfèrent au même emplacement.

```
if ($ref1 == $ref2) {
 print "refs 1 et 2 font référence à la même chose\n";
}
```

L'utilisation d'une référence en tant que chaîne produit à la fois le type d'objet qu'elle référence en incluant le nom du paquetage l'ayant éventuellement consacré (par `bless()`) comme expliqué dans *perlobj*, et aussi son adresse mémoire numérique en hexadécimal. L'opérateur `ref()` produit juste le type d'objet lié à la référence, sans l'adresse. Voir `ref` in *perlfunc* pour plus de détails et des exemples d'utilisation.

L'opérateur `bless()` peut être utilisé pour associer l'objet, sur lequel pointe une référence, avec un paquetage fonctionnant comme une classe d'objets. Cf. *perlobj*.

Un typeglob peut être déréféré de la même façon qu'une référence, car la syntaxe de déréférencement indique toujours le type de référence souhaité. Par conséquent, `${*foo}` et `${\foo}` indique tous les deux la même variable scalaire.

Voici un truc pour interpoler l'appel d'une routine dans une chaîne de caractères :

```
print "My sub returned @{$mysub(1,2,3)} that time.\n";
```

La façon dont ça marche, c'est que lorsque le `@{...}` est aperçu à l'intérieur des guillemets de la chaîne de caractères, il est évalué comme un bloc. Le bloc crée une référence à une tableau anonyme contenant le résultat de l'appel à `mysub(1,2,3)`. Le bloc entier retourne ainsi une référence à un tableau, qui est alors déréféré par `@{...}` et inséré dans la chaîne de caractères entre guillemets. Cette chipotterie est aussi utile pour des expressions arbitraires :

```
print "That yields @{$n + 5} widgets\n";
```

### 40.2.3 Références symboliques

Nous avons déjà expliqué que, quand c'est nécessaire, les références devaient exister si elles sont définies, mais nous n'avons pas dit ce qui arrivait lorsqu'une valeur utilisée comme référence est déjà définie mais n'est *pas* une référence dure. Si vous l'utilisez comme référence dans ce cas-là, elle sera traitée comme une référence symbolique. C'est-à-dire que la valeur du scalaire est considérée comme le *nom* d'une variable, plutôt que comme un lien direct vers une (éventuelle) valeur anonyme.

En général, les gens s'attendent à ce que ça fonctionne de cette façon. C'est donc comme ça que ça marche.

```
$name = "foo";
$$name = 1; # Affecte $foo
${$name} = 2; # Affecte $foo
${$name x 2} = 3; # Affecte $foofoo
$name->[0] = 4; # Affecte $foo[0]
@$name = (); # Efface @foo
&$name(); # Appelle &foo() (comme en Perl 4)
$pack = "THAT";
"${$pack}::$name" = 5; # Affecte $THAT::foo sans évaluation
```

C'est très puissant, et potentiellement dangereux, dans le sens où il est possible de vouloir (avec la plus grande sincérité) utiliser une référence dure, et utiliser accidentellement une référence symbolique à la place. Pour vous en prémunir, vous pouvez utiliser

```
use strict 'refs';
```

et seules les références dures seront alors autorisées dans le reste du bloc l'incluant. Un bloc imbriqué peut inverser son effet avec

```
no strict 'refs';
```



Seuls les variables (globales, même si elles sont localisées) de paquetage sont visibles par des références symboliques. Les variables lexicales (déclarées avec `my()`) ne font pas partie de la table des symboles, et sont donc invisibles à ce mécanisme. Par exemple :

```
local $value = 10;
$ref = "value";
{
 my $value = 20;
 print $$ref;
}
```

Ceci imprimera 10 et non pas 20. Souvenez-vous que `local()` affecte les variables de paquetage, qui sont toutes "globales" au paquetage.

#### 40.2.4 Références pas-si-symboliques-que-ça

Une nouvelle fonctionnalité contribuant à la lisibilité en perl version 5.001 est que les crochets autour d'une référence symbolique se comportent comme des apostrophes, tout comme elles l'ont toujours été dans une chaîne de caractères. C'est-à-dire que

```
$push = "pop on ";
print "${push}over";
```

a toujours imprimé "pop on over", même si `push` est un mot réservé. Ceci a été généralisé pour fonctionner de même en dehors de guillemets, de telle sorte que

```
print ${push} . "over";
```

et même

```
print ${ push } . "over";
```

auront un effet identique. (Ceci aurait provoqué une erreur syntaxique en Perl 5.000, bien que Perl 4 l'autorisait dans une forme sans espaces.) Cette construction n'est *pas* considérée comme une référence symbolique lorsque vous utilisez `strict refs` :

```
use strict 'refs';
${ bareword }; # Correct, signifie $bareword.
${ "bareword" }; # Erreur, référence symbolique.
```

De façon similaire, à cause de tout le subscripting qui est effectué en utilisant des mots simples, nous avons appliqué la même règle à tout mot simple qui soit utilisé pour le subscripting d'une table de hachage. Désormais, au lieu d'écrire

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

vous pourrez donc juste écrire

```
$array{ aaa }{ bbb }{ ccc }
```

sans vous inquiéter du fait que les subscripts soient ou non des mots réservés. Dans les rares cas où vous souhaiteriez faire quelque chose comme :

```
$array{ shift }
```

vous pouvez en forcer l'interprétation comme un mot réservé en ajoutant n'importe quoi qui soit plus qu'un mot simple :

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

La directive `use warnings` ou l'option `-w` vous avertira si un mot réservé est interprété comme une chaîne de caractères. Mais il ne vous avertira plus si vous utilisez des mots en minuscules, car la chaîne de caractères est entre guillemets de façon effective.

### 40.2.5 Pseudo-tables de hachage : utiliser un tableau comme table de hachage

**AVERTISSEMENT** : cette section traite de fonctionnalités expérimentales. Certains détails pourraient changer sans annonce particulière dans les prochaines versions.

**NOTE** : la partie visible de l'implémentation actuelle des pseudo-tables de hachage (l'utilisation singulière du premier élément du tableau) est dépréciée à partir de Perl 5.8.0 et disparaîtra dans Perl 5.10.0. Cette fonctionnalité sera implémentée autrement. Au-delà de l'interface actuelle qui est particulièrement horrible, l'implémentation actuelle ralentit notablement l'utilisation normale des tableaux et des tables de hachage. La directive `'fields'` restera disponible.

Avec la version 5.005 de Perl, vous pouvez désormais utiliser une référence à un tableau dans un contexte qui exigerait normalement une référence à une table de hachage. Ceci vous permet d'accéder aux éléments d'un tableau en utilisant des noms symboliques, comme s'ils étaient les champs d'une structure.

Pour que cela fonctionne, le tableau doit contenir des informations supplémentaires. Le premier élément du tableau doit être une référence à une table de hachage qui associe les noms de champs avec les indices du tableau. Voici un exemple :

```
$struct = [{foo => 1, bar => 2}, "FOO", "BAR"];

$struct->{foo}; # identique à $struct->[1], c'est-à-dire "FOO"
$struct->{bar}; # identique à $struct->[2], c'est-à-dire "BAR"

keys %$struct; # retournera ("foo", "bar") dans un certain ordre
values %$struct; # retournera ("FOO", "BAR") dans le meme certain ordre

while (my($k,$v) = each %$struct) {
 print "$k => $v\n";
}
```

Perl déclenchera une exception si vous essayez d'accéder à des champs inexistants. Pour éviter les incohérences, utilisez toujours la fonction `fields::phash()` fournie par la directive `fields`.

```
use fields;
$pseudohash = fields::phash(foo => "FOO", bar => "BAR");
```

Pour de meilleures performances, Perl peut aussi effectuer la traduction des noms de champs en indices de tableau lors de la compilation pour les références à des objets typées. Voir *fields*.

Il existe deux moyens de vérifier l'existence d'une clé dans une pseudo-table de hachage. Le premier est d'utiliser `exists()`. Cela teste si ce champ donné a déjà été utilisé. Ce comportement est le même que celui d'une table de hachage normale. Par exemple :

```
use fields;
$phash = fields::phash([qw(foo bar pants)], ['FOO']);
$phash->{pants} = undef;

print exists $phash->{foo}; # vrai, 'foo' est valué dans la declaration
print exists $phash->{bar}; # faux, 'bar' n'a jamais été utilisé
print exists $phash->{pants}; # vrai, 'pants' a été utilisé
```

Le second moyen est d'utiliser `exists()` sur la table de hachage présente comme premier élément du tableau. Cela teste si ce champ est un champ valide pour cette pseudo-table de hachage.

```
print exists $phash->[0]{bar}; # vrai, 'bar' est valide
print exists $phash->[0]{shoes}; # faux, 'shoes' ne peut être utilisé
```

Un appel à `delete()` sur un élément d'une pseudo-table de hachage n'efface que le valeur correspondant à cette clé et non la clé elle-même. Pour effacer la clé, vous devez l'effacer explicitement de la table située au premier élément du tableau.

```
print delete $phash->{foo}; # affiche $phash->[1], "FOO"
print exists $phash->{foo}; # faux
print exists $phash->[0]{foo}; # vrai, la clé existe encore
print delete $phash->[0]{foo}; # maintenant la clé est effacée
print $phash->{foo}; # exception déclenchée
```

### 40.2.6 Modèles de fonctions

Comme expliqué ci-dessus, une fermeture est une fonction anonyme qui a accès aux variables lexicales qui étaient visibles lors de sa compilation. Elle conserve l'accès à ses variables même si elle n'est exécutée que plus tard, comme dans le cas des signaux ou des callbacks en Tk.

Utiliser une fermeture comme modèle de fonction nous permet de générer de nombreuses fonctions agissant de façon similaire. Supposons que vous souhaitiez des fonctions nommées d'après la couleur qu'elles produiront en HTML via la balise FONT :

```
print "Be ", red("careful"), "with that ", green("light");
```

Les fonctions red() et green() seront très similaires. Pour les créer, nous allons assigner une fermeture à un typeglob du nom de la fonction que nous voulonsq construire.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
 no strict 'refs'; # Autorise la manipulation de la table de symboles
 *$name = *{uc $name} = sub { "@_" };
}
```

Désormais, toutes ces fonctions existent de façon indépendante. Vous pouvez appeler red(), RED(), blue(), BLUE(), green(), etc. Cette technique optimise le temps de compilation et l'utilisation de la mémoire, et elle est aussi moins sujette aux erreurs puisque la vérification syntaxique a lieu à la compilation. Il est nécessaire qu'aucune variable de la routine anonyme ne soit lexicale pour créer une fermeture propre. C'est la raison pour laquelle nous avons un my dans notre boucle.

C'est l'un des seuls endroits où donner un prototype à une fermeture a un réel sens. Si vous souhaitiez imposer un contexte scalaire aux arguments de ces fonctions (ce qui n'est probablement pas une bonne idée pour cet exemple particulier), vous auriez pu écrire à la place :

```
*$name = sub ($) { "$_[0]" };
```

Quoi qu'il en soit, comme la vérification des prototypes a lieu à la compilation, l'affectation ci-dessus est effectuée trop tard pour être vraiment utile. Vous pourriez gérer ça en insérant la boucle entière d'affectations dans un bloc BEGIN, forçant ainsi son exécution pendant la compilation.

L'accès aux lexicaux qui change au-delà des types (comme ceux de la boucle for ci-dessus) ne fonctionne qu'avec des fermetures et pas avec des routines générales. Par conséquent, dans le cas général, les routines nommées ne s'imbriquent pas proprement, au contraire des routines anonymes. C'est comme cela parce que les routines nommées sont créées (et récupèrent les lexicaux externes) une seule fois lors de la compilation alors que les routines anonymes réalisent cette récupération à chaque exécution de l'opérateur 'sub'. Si vous êtes habitué à l'utilisation de routines imbriquées dans d'autres langages de programmation, avec leurs propres variables privées, il va vous falloir travailler là-dessus en Perl un tant soit peu. La programmation intuitive de ce genre de choses implique des avertissements mystérieux du genre "will not stay shared". Par exemple, ceci ne fonctionnera pas :

```
sub outer {
 my $x = $_[0] + 35;
 sub inner { return $x * 19 } # FAUX
 return $x + inner();
}
```

Une solution pourrait être celle-ci :

```
sub outer {
 my $x = $_[0] + 35;
 local *inner = sub { return $x * 19 };
 return $x + inner();
}
```

Maintenant, inner() ne peut être appelée que de l'intérieur de outer(), grâce aux affectations temporaires de la fermeture (routine anonyme). Mais lorsque cela a lieu, elle a un accès normal à la variable lexicale \$x dans la portée de outer().

Ceci a pour effet intéressant de créer une fonction locale à une autre, ce qui n'est pas normalement supporté par Perl.

## 40.3 AVERTISSEMENT

Vous ne devriez pas (utilement) utiliser une référence comme clé d'une table de hachage. Elle sera convertie en chaîne de caractères :

```
$x{ \$a } = $a;
```

Si vous essayez de déréférencer la clé, il n'y aura pas de déréférencement dur et vous ne ferez pas ce que vous souhaitez. Vous devriez plutôt faire ainsi :

```
$r = \@a;
$x{ $r } = $r;
```

Et alors, au moins, vous pourrez utiliser les valeurs, par `values()`, qui seront de véritables références, au contraire des clés, par `keys()`.

Le module standard `Tie::RefHash` fournit une base de travail pratique pour faire ce genre de choses.

## 40.4 VOIR AUSSI

À côté de la documentation standard, du code source peut être instructif. Quelques exemples pathologiques de l'utilisation de références peuvent être trouvées dans le test de régression *t/op/ref.t* du répertoire source de Perl.

Voir aussi *perldsc* et *perllol*, pour l'utilisation de références dans la création de structures de données complexes, et *perltot*, *perlobj* et *perlbot* pour leur utilisation dans la création d'objets.

## 40.5 AUTEUR

Larry Wall

## 40.6 TRADUCTION

### 40.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 40.6.2 Traducteur

Traduction initiale : Jean-Pascal Peltier <jp\_peltier@altavista.net>. Mise à jour : Paul Gaborit <paul.gaborit@enstimac.fr>.

### 40.6.3 Relecture

Personne pour l'instant.

# Chapitre 41

## perform

Formats Perl

### 41.1 DESCRIPTION

Perl possède un mécanisme qui permet de générer des rapports et tableaux simples. Pour ce faire, il vous aide à écrire le code de manière semblable à ce à quoi il ressemblera lors de l'impression. On peut garder la trace du nombre de lignes par page, sur quelle page on se trouve, quand imprimer des entêtes, etc. Les mots-clés sont empruntés au FORTRAN : `format()` pour déclarer, `write()` pour exécuter ; référez-vous à leurs entrées dans *perlfunc*. Heureusement, le layout est largement plus lisible, un peu comme l'instruction `PRINT USING` du BASIC. Voyez-le comme une sorte de « `nroff(1)` du pauvre ».

Les formats, comme les paquetages et les sous-programmes, sont déclarés plutôt qu'exécutés : ils peuvent donc apparaître à n'importe quel point de vos programmes, mais mieux vaut les regrouper. Ils ont leur propre espace de nommage, bien séparé des autres « types » de Perl. Cela signifie que peuvent coexister une fonction « bidule » et un format « bidule ». Cependant, le nom par défaut d'un format associé à un fichier est le nom du fichier. Par conséquent, le nom par défaut du format pour `STDOUT` est « `STDOUT` », et le nom du format par défaut pour le fichier `TEMP` est « `TEMP` ». Ils ont l'air semblables, mais ne le sont pas.

Les formats de sortie sont déclarés comme suit :

```
format NOM =
LISTEDEFORMATS
.
```

Si le nom est omis, le format «`STDOUT`» est alors automatiquement défini. `LISTEDEFORMATS` consiste en une suite de lignes, chacune d'elle pouvant être de l'un des trois types suivants :

1. Un commentaire, indiqué par un '#' dans la première colonne.
2. Une ligne-image donnant le format de la ligne.
3. Une ligne d'arguments, donnant les valeurs à insérer dans la ligne-image précédente.

Les lignes-images s'imprimeront exactement comme elles ont été écrites, à l'exception des champs remplacés par des arguments dans ces lignes. Chaque champ d'une ligne-image commence par une arrobe (@) ou un accent circonflexe (^). Ces lignes ne font l'objet d'aucune interprétation. Le champ @, à ne pas confondre avec le symbole de tableau « @ », est le type de champ normal. L'autre type, le champ ^, sert à faire du remplissage multiligne rudimentaire. On définit la longueur de champ en le remplissant avec les caractères « < », « > », ou « | », pour aligner, respectivement, à gauche, à droite, ou au centre. Si la variable excède la longueur spécifiée, elle sera tronquée.

On peut aussi aligner à droite en utilisant le caractère « # », avec un « . » optionnel, pour spécifier un champ numérique : l'alignement se fait sur le « . » décimal. Si la valeur spécifiée pour ces types de champs contient un retour à la ligne, seul le texte jusqu'au retour à la ligne est imprimé. Enfin, le champ spécial « @\* » peut être employé pour écrire des valeurs multilignes non tronquées ; il doit apparaître seul sur la ligne.

Les valeurs sont spécifiées sur la ligne suivante, dans le même ordre que les champs images. Les expressions fournissant les valeurs doivent être séparées par des virgules. Les expressions sont toutes évaluées en tant que liste avant que la ligne ne soit traitée. Une seule expression peut donc créer toute une liste d'éléments. Les expressions peuvent être écrites sur plusieurs lignes, à condition de les placer entre parenthèses. En pareil cas, la parenthèse ouvrante *doit* commencer la









### 41.2.2 Accéder aux formats de l'intérieur

Pour un accès de bas niveau au mécanisme de formatage, vous pouvez utiliser `formline()` et accéder à la variable `^A` (la variable `$ACCUMULATOR`) directement.

Par exemple :

```
$str = formline <<'END', 1,2,3;
@<<< @||| @>>>
END

print "Yo, je viens de mettre '^A' dans l'accumulateur !\n";
```

Ou faire une routine `swrite()`, qui est à `write()` ce que `sprintf()` est à `printf()`, comme suit :

```
use Carp;
sub swrite {
 croak "utilisation : swrite IMAGE ARGUMENTS" unless @_;
 my $format = shift;
 $^A = "";
 formline($format,@_);
 return $^A;
}

$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<< @||| @>>>
END
print $string;
```

## 41.3 MISE EN GARDE

Le point isolé qui termine un format peut aussi provoquer la perte d'un courriel passant via un serveur de mail mal configuré (et l'expérience montre qu'une telle configuration est la règle, et non pas l'exception). Donc, lorsque vous envoyez un format par courriel, indentez-le ! ainsi, le point terminant le format ne soit pas aligné à gauche : cela évitera une coupe par le serveur SMTP.

Les variables lexicales (déclarées avec « `my` ») ne sont pas visibles dans un format sauf celui-ci est déclaré dans le champ de vision de la variable lexicale. Ils n'étaient pas visibles du tout avant la version 5.001.

Les formats sont le seul morceau de Perl qui utilisent de façon inconditionnelle les informations provenant de la locale d'un programme. Si l'environnement du programme spécifie une locale `LC_NUMERIC`, elle sera toujours utilisée pour spécifier le point décimal dans une sortie formatée. Perl ignore tout simplement le reste de la locale si le pragma `use locale` n'est pas utilisé. Les sorties formatées ne peuvent pas prendre en compte ce pragma car il est lié à la structure de bloc du programme, et, pour des raisons historiques, les formats sont définis hors de cette structure. Référez-vous à *perllocale* pour plus d'informations sur la gestion des locales.

## 41.4 TRADUCTION

### 41.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 41.4.2 Traducteur

Mathieu Arnold <arn\_mat@club-internet.fr>.

### 41.4.3 Relecture

Yves Maniette <yves@giga.sct.ub.es>, Gérard Delafond.

# Chapitre 42

## perlobj

Objets en Perl

### 42.1 DESCRIPTION

Tout d'abord, vous devez comprendre ce que sont les références en Perl. Voir *perlref* pour cela. Ensuite, si le document qui suit vous semble encore trop compliqué, vous trouverez des tutoriels sur la programmation orientée objet en Perl dans *perltoot* et *perltooc*.

Si vous êtes toujours avec nous, voici trois définitions très simples que vous devriez trouver rassurantes.

1. Un objet est simplement une référence qui sait à quelle classe elle appartient.
2. Une classe est simplement un paquetage qui fournit des méthodes pour manipuler les références d'objet.
3. Une méthode est simplement un sous-programme qui attend une référence d'objet (ou un nom de paquetage, pour les méthodes de classe) comme premier argument.

Nous allons maintenant couvrir ces points plus en détails.

#### 42.1.1 Un objet est simplement une référence

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe particulière pour les constructeurs. Un constructeur est juste un sous-programme qui retourne une référence à quelque chose qui a été "consacré" (ou "béni") par une classe, généralement la classe dans laquelle le sous-programme est défini. Voici un constructeur typique :

```
package Critter;
sub new { bless {} }
```

Le mot `new` n'a rien de spécial. Vous auriez aussi bien pu écrire un constructeur de cette façon :

```
package Critter;
sub spawn { bless {} }
```

Ceci peut même être préférable car les programmeurs C++ n'auront pas tendance à penser que `new` fonctionne en Perl de la même manière qu'en C++. Ce n'est pas le cas. Nous vous recommandons de nommer vos constructeurs de façon qu'ils aient un sens en fonction du contexte du problème que vous résolvez. Par exemple, les constructeurs dans l'extension Tk de Perl portent les noms des widgets qu'ils créent.

Une différence entre les constructeurs de Perl et de C++ est qu'en Perl, ils doivent allouer leur propre mémoire (l'autre différence est qu'ils n'appellent pas automatiquement les constructeurs de classe de base surchargés). Le `{}` alloue un hachage anonyme ne contenant aucune paire clé/valeur, et le retourne. Le `bless()` prend cette référence, dit à l'objet qu'il référence qu'il est désormais un Critter et retourne la référence. C'est pour que cela soit plus pratique, car l'objet référencé sait lui-même qu'il a été consacré, et sa référence aurait pu être retournée directement, comme ceci :

```
sub new {
 my $self = {};
 bless $self;
 return $self;
}
```

Vous voyez souvent de telles choses dans des constructeurs plus compliqués qui veulent utiliser des méthodes de la classe pour la construction :

```
sub new {
 my $self = {};
 bless $self;
 $self->initialize();
 return $self;
}
```

Si vous vous souciez de l'héritage (et vous devriez ; voir *Modules: création, utilisation et abus in perlmodlib*), alors vous préférerez utiliser la forme à deux arguments de `bless` pour que vos constructeurs puissent être utilisés par héritage :

```
sub new {
 my $class = shift;
 my $self = {};
 bless $self, $class;
 $self->initialize();
 return $self;
}
```

Ou si vous vous attendez à ce que les gens appellent non seulement `CLASS->new()`, mais aussi `$obj->new()`, alors utilisez quelque chose comme ce qui suit (notez que cet appel à `new()` via une instance ne réalise aucune copie automatiquement. Que vous vouliez une copie superficielle ou en profondeur, dans tous les cas vous aurez à écrire le code correspondant). La méthode `initialize()` sera celle de la classe dans laquelle nous consacrons l'objet :

```
sub new {
 my $this = shift;
 my $class = ref($this) || $this;
 my $self = {};
 bless $self, $class;
 $self->initialize();
 return $self;
}
```

À l'intérieur du paquetage de la classe, les méthodes géreront habituellement la référence comme une référence ordinaire. À l'extérieur du paquetage, la référence est généralement traitée comme une valeur opaque à laquelle on ne peut accéder qu'à travers les méthodes de la classe.

Bien qu'un constructeur puisse en théorie re-consacrer un objet référencé appartenant couramment à une autre classe, ceci va presque certainement vous causer des problèmes. La nouvelle classe est responsable de tout le nettoyage qui viendra plus tard. La précédente consécration est oubliée, puisqu'un objet ne peut appartenir qu'à une seule classe à la fois (même si bien sûr il est libre d'hériter de méthodes en provenance de nombreuses classes). Si toutefois vous vous retrouvez dans l'obligation de le faire, la classe parent a probablement un mauvais comportement.

Une clarification : les objets de Perl sont consacrés. Les références ne le sont pas. Les objets savent à quel paquetage ils appartiennent. Pas les références. La fonction `bless()` utilise la référence pour trouver l'objet. Considérez l'exemple suivant :

```
$a = {};
$b = $a;
bless $a, BLAH;
print "\$b is a ", ref($b), "\n";
```

Ceci rapporte `$b` comme étant un `BLAH`, il est donc évident que `bless()` a agi sur l'objet et pas sur la référence.

### 42.1.2 Une classe est simplement un paquetage

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe spéciale pour les définitions de classes. Vous utilisez un paquetage en tant que classe en mettant des définitions de méthodes dans la classe.

Il existe un tableau spécial appelé @ISA à l'intérieur de chaque paquetage, qui dit où trouver une méthode si on ne la trouve pas dans le paquetage courant. C'est de cette façon que Perl implémente l'héritage. Chaque élément du tableau @ISA est juste le nom d'un autre paquetage qui s'avère être un paquetage de classe. Les méthodes manquantes sont recherchées dans cette arborescence de classes en profondeur et de gauche à droite par défaut (voir *mro* pour spécifier d'autres ordres de recherche). Les classes accessibles à travers @ISA sont les classes de base de la classe courante.

Toutes les classes héritent implicitement de la classe UNIVERSAL en tant que dernière classe de base. Plusieurs méthodes couramment utilisées sont automatiquement fournies par la classe UNIVERSAL ; voir Méthodes UNIVERSAL par défaut (§42.1.6) pour plus de détails.

Si une méthode manquante est trouvée dans une classe de base, elle est mise en cache dans la classe courante pour plus d'efficacité. Modifier @ISA ou définir de nouveaux sous-programmes invalide le cache et force Perl à recommencer la recherche.

Si ni la classe courante, ni ses classes de base nommées, ni la classe UNIVERSAL ne contiennent la méthode requise, ces trois endroits sont fouillés de nouveau, cette fois à la recherche d'une méthode appelée AUTOLOAD(). Si une méthode AUTOLOAD est trouvée, cette méthode est appelée à la place de la méthode manquante et le nom complet de la méthode qui devait être appelée est stocké dans la variable globale de paquetage \$AUTOLOAD.

Si rien de tout cela ne marche, Perl abandonne finalement et se plaint.

Si vous voulez stopper l'héritage par AUTOLOAD à votre niveau, il vous suffit de dire :

```
sub AUTOLOAD;
```

et l'appel mourra via die en utilisant le nom de la méthode appelée.

Les classes de Perl ne font que de l'héritage de méthodes. L'héritage de données est laissé à la charge de la classe elle-même. Ce n'est pas, et de loin, un problème en Perl car la plupart des classes stockent les attributs de leurs objets dans un hachage anonyme qu'elles utilisent comme un espace de nommage qui leur est propre mais qui peut être ciselé par les diverses autres classes qui veulent faire quelque chose de l'objet. Le seul problème dans ce cas est que vous ne pouvez pas être certain que vous n'utilisez pas un morceau du hachage qui serait déjà utilisé par ailleurs. Une façon raisonnable de le contourner est de préfixer vos noms d'attributs par le nom de votre paquetage.

```
sub bump {
 my $self = shift;
 $self->{ __PACKAGE__ . ".count" }++;
}
```

### 42.1.3 Une méthode est simplement un sous-programme

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe spéciale pour la définition des méthodes (il fournit toutefois un peu de syntaxe pour l'invocation des méthodes. Vous en saurez plus à ce sujet plus tard). Une méthode s'attend à ce que son premier argument soit l'objet (référence) ou le paquetage (chaîne) pour lequel elle est invoquée. Il existe deux façons d'appeler les méthodes, que nous appellerons des méthodes de classe et des méthodes d'instance.

Une méthode de classe attend un nom de classe comme premier argument. Elle fournit une fonctionnalité à la classe toute entière mais pas à un objet en particulier appartenant à cette classe. Les constructeurs sont souvent des méthodes de classe mais voyez *perltoot* et *perltooc* pour des alternatives. De nombreuses méthodes de classe ignorent tout simplement leur premier argument car elles savent déjà dans quel paquetage elles sont, et se moquent du paquetage via lequel elles ont été invoquées (ce ne sont pas nécessairement les mêmes, car les méthodes de classe suivent l'arbre d'héritage tout comme les méthodes d'instance ordinaires). Un autre usage typique des méthodes de classe est la recherche d'un objet par son nom :

```
sub find {
 my ($class, $name) = @_ ;
 $objtable{$name};
}
```

Une méthode d'instanciation attend une référence à un objet comme premier argument. Typiquement, elle change le premier argument en variable "self" ou "this", puis l'utilise comme une référence ordinaire.

```

sub display {
 my $self = shift;
 my @keys = @_ ? @_ : sort keys %$self;
 foreach $key (@keys) {
 print "\t$key => $self->{$key}\n";
 }
}

```

#### 42.1.4 Invocation de méthode

Pour des raisons historiques et autres, Perl offre deux moyens équivalents d'appeler des méthodes. Le plus simple et le plus courant est la notation à base de flèche :

```

my $fred = Critter->find("Fred");
$fred->display("Height", "Weight");

```

L'usage de la flèche avec des références doit déjà vous être familier. En fait, comme \$fred fait référence à un objet, vous pouvez considérer l'appel à la méthode comme une autre forme de déréférencement.

Quoiqu'il y ait à gauche de la flèche, que ce soit une référence ou un nom de classe, c'est ce qui sera passé à la méthode comme premier argument. Donc le code ci-dessus est quasiment équivalent à :

```

my $fred = Critter::find("Critter", "Fred");
Critter::display($fred, "Height", "Weight");

```

Comment Perl peut-il savoir dans quel paquetage est la méthode ? En regardant la partie gauche de la flèche, qui doit être soit une référence à un objet soit un nom de classe, c'est-à-dire quelque chose qui a été consacré par un paquetage. C'est à partir de ce paquetage que Perl commence la recherche. Si ce paquetage ne propose pas cette méthode, Perl cherche dans les classes de base de ce paquetage et ainsi de suite.

Si besoin est, vous *pouvez* forcer Perl à commencer sa recherche dans un autre paquetage.

```

my $barney = MyCritter->Critter::find("Barney");
$barney->Critter::display("Height", "Weight");

```

Dans cet exemple MyCritter est a priori une sous-classe de Critter qui définit ses propres versions de find() et de display(). Nous ne les avons pas spécifiées mais cela n'a pas d'importance puisqu'ici nous forçons Perl à commencer sa recherche de sous-routines dans Critter.

Un cas spécial de la situation précédente est l'utilisation de la pseudo classe SUPER pour demander à Perl d'effectuer la recherche de méthodes dans les paquetages de la liste @ISA de la classe courante.

```

package MyCritter;
use base 'Critter'; # sets @MyCritter::ISA = ('Critter');

sub display {
 my ($self, @args) = @_;
 $self->SUPER::display("Name", @args);
}

```

Il est important de noter que SUPER se réfère à la (aux) superclasse(s) du *paquetage courant* et non à la (aux) superclasse(s) de l'objet lui-même. De plus, la pseudo classe SUPER peut être utilisée comme modificateur d'un nom de méthode mais pas aux autres endroits où un nom de classe est utilisé. Exemple :

```

something->SUPER::method(...); # OK
SUPER::method(...); # MAUVAIS
SUPER->method(...); # MAUVAIS

```

À la place d'un nom de classe ou d'une référence à un objet, vous pouvez utiliser n'importe quelle expression qui retourne quelque chose pouvant apparaître à gauche de la flèche. Donc, l'instruction suivante est valide :

```

Critter->find("Fred")->display("Height", "Weight");

```

et celle-ci aussi :

```

my $fred = (reverse "rettirC")->find(reverse "derF");

```

À droite de la flèche, on trouve habituellement le nom de la méthode mais une simple variable scalaire contenant soit le nom de la méthode soit une référence à un sous-programme peut très bien convenir.

### 42.1.5 Syntaxe objet indirecte

Une autre manière d'appeler une méthode passe par la notation indirecte. Cette syntaxe était utilisée dans Perl 4 bien avant l'introduction des objets et sert encore avec les handle de fichiers comme dans :

```
print STDERR "help!!!\n";
```

Cette même syntaxe peut être utilisée pour appeler des méthodes de classe ou d'instance.

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

Notez bien l'absence de virgule entre l'objet ou le nom de classe et les paramètres. C'est cela qui indique Perl que vous voulez faire appel à une méthode plutôt qu'un classique appel de sous-routine.

Mais que se passe-t-il s'il n'y a pas de paramètres ? Dans ce cas Perl doit deviner ce que vous voulez faire. De plus, il doit le savoir *lors de la compilation*. Dans la plupart des cas, Perl devine correctement mais s'il se trompe, vous vous retrouvez avec un appel de méthode à la place d'un appel de fonction ou vice-versa. Cela introduit de bogues subtils qu'il est difficile de détecter.

Par exemple, l'appel à la méthode `new` en notation indirecte – comme les programmeurs C++ ont l'habitude de le faire – peut être compilé de manière erronée en un appel de sous-routine s'il existe une fonction `new` dans la portée de l'appel. Cela se termine par l'appel de la sous-routine `new` du paquetage courant plutôt que par la méthode de la classe voulue. Le compilateur tente de tricher en se souvenant des noms employés par des `require` mais le petit gain attendu ne vaut pas les années de débogage nécessaires lorsqu'il se trompe.

Il y a un autre problème avec cette syntaxe : l'objet indirect est limité à un nom, une variable scalaire ou un bloc, pour éviter de regarder trop loin en avant. (Ces mêmes règles bizarres sont utilisées pour l'emplacement du handle de fichier dans les fonctions telles que `print` et `printf`) Ceci peut mener à des problèmes de précedence horriblement troublants, comme dans ces deux lignes :

```
move $obj->{FIELD}; # probablement mauvaise !
move $ary[$i]; # probablement mauvaise !
```

qui, étonnamment, sont interprétées comme ceci :

```
$obj->move->{FIELD}; # Étonnant...
$ary->move([$i]); # Vous ne vous y attendiez pas !
```

plutôt que comme cela :

```
$obj->{FIELD}->move(); # Vous seriez chanceux
$ary[$i]->move; # ...
```

Pour obtenir le comportement correct avec la notation indirecte, vous pourriez utiliser un bloc autour de l'objet indirect :

```
move {$obj->{FIELD}};
move {$ary[$i]};
```

Hélas, vous aurez encore une ambiguïté s'il existe une fonction nommée `move` dans le paquetage courant. **La notation `->` suffit pour lever toutes ces ambiguïtés. Nous vous recommandons donc de l'utiliser en toutes circonstances.** En revanche, il peut encore arriver que vous ayez à lire du code utilisant la notation indirecte. Il est donc important que vous soyez familiariser avec elle.

### 42.1.6 Méthodes UNIVERSAL par défaut

Le paquetage UNIVERSAL contient automatiquement les méthodes suivantes qui sont héritées par toutes les autres classes :

#### isa(CLASSE)

`isa` retourne *vrai* si son objet est consacré par une sous-classe de CLASSE.

Vous pouvez aussi appeler `UNIVERSAL::isa` comme une simple fonction avec deux arguments. Bien sûr, cela ne fonctionnera pas si quelqu'un redéfinit `isa` dans une classe, ce qui n'est donc pas une chose à faire.

Pour vérifier que ce que vous recevez est correct, utilisez la fonction `blessed` du module `Scalar::Util` :

```
if(blessed($ref) && $ref->isa('Une::Classe')) {
 #...
}
```

`blessed` retourne le nom du paquetage ayant consacré son argument (ou `undef`).

#### can(METHODE)

`can` vérifie si son objet possède une méthode appelée METHODE, si c'est le cas, une référence à la routine est retournée, sinon c'est *undef* qui est renvoyé.

`UNIVERSAL::can` peut aussi être appelé comme une subroutine à deux arguments. Elle retourne toujours *undef* si son premier argument n'est pas un objet ou le nom d'une classe. Les mêmes conditions d'appel que celles de `UNIVERSAL::isa` s'appliquent.

#### VERSION( [NEED] )

`VERSION` retourne le numéro de version de la classe (du paquetage). Si l'argument `NEED` est fourni, elle vérifie que le numéro de version courant (tel que défini par la variable `$VERSION` dans le paquetage donné) n'est pas inférieur à `NEED` ; il mourra si ce n'est pas le cas. Cette méthode est normalement appelée en tant que méthode de classe. Elle est appelée automatiquement par la forme `VERSION` de `use`.

```
use A 1.2 qw(des routines importees);
qui implique :
A->VERSION(1.2);
```

**NOTE :** `can` utilise directement le code interne de Perl pour la recherche de méthode, et `isa` utilise une méthode très similaire et une stratégie de cache. Ceci peut produire des effets étranges si le code Perl change dynamiquement `@ISA` dans un paquetage.

Vous pouvez ajouter d'autres méthodes à la classe UNIVERSAL via du code Perl ou XS. Vous n'avez pas besoin de préciser `use UNIVERSAL` (et vous ne devriez pas le faire) pour que ces méthodes soient disponibles dans votre programme.

### 42.1.7 Destructeurs

Lorsque la dernière référence à un objet disparaît, l'objet est automatiquement détruit (Ce qui peut même se produire après un `exit()` si vous avez stocké des références dans des variables globales). Si vous voulez prendre le contrôle juste avant que l'objet ne soit libéré, vous pouvez définir une méthode `DESTROY` dans votre classe. Elle sera appelée automatiquement au moment approprié, et vous pourrez y réaliser tous les nettoyages supplémentaires dont vous avez besoin. Perl passe une référence à l'objet qui va être détruit comme premier (et unique) argument. Souvenez-vous que cette référence est une valeur en lecture seule, qui ne peut pas être modifiée en manipulant `$_[0]` au sein du destructeur. L'objet en lui-même (i.e. le bidule vers lequel pointe la référence, appelé `$_[0]`, `@{$_[0]}`, `%{$_[0]}` etc.) n'est pas soumis à la même contrainte.

Puisque les méthodes `DESTROY` peuvent être appelées n'importe quand, il est important de rendre locale toute variable globale utilisée. En particulier, localisez `$@` si vous utiliser `eval {}` et localisez `$?` si vous utiliser `system` ou les apostrophes inverses.

Si vous vous arrangez pour re-consacrer la référence avant la fin du destructeur, Perl appellera de nouveau la méthode `DESTROY` après la fin de l'appel en cours pour l'objet re-consacré. Cela peut être utilisé pour une délégation propre de la destruction d'objet, ou pour s'assurer que les destructeurs dans la classe de base de votre choix sont appelés. L'appel explicite de `DESTROY` est aussi possible, mais n'est habituellement pas nécessaire.

Ne confondez pas ce qui précède avec la façon dont sont détruits les objets *CONTENUS* dans l'objet courant. De tels objets seront libérés et détruits automatiquement en même temps que l'objet courant, pourvu qu'il n'existe pas ailleurs d'autres références pointant vers eux.

### 42.1.8 Résumé

C'est à peu près tout sur le sujet. Il ne vous reste plus qu'à aller acheter un livre sur la méthodologie de conception orientée objet, et vous le frapper sur le front pendant les six prochains mois environ.

### 42.1.9 Ramasse-miettes à deux phases

Dans la plupart des cas, Perl utilise un système de ramasse-miettes simple et rapide basé sur les références. Cela signifie qu'il se produit un déréférencement supplémentaire à un certain niveau, donc si vous n'avez pas compilé votre exécutable de Perl en utilisant l'option `-O` de votre compilateur C, les performances s'en ressentiront. Si vous avez compilé Perl avec `cc -O`, cela ne comptera probablement pas.

Un souci plus sérieux est que la mémoire inaccessible avec un compteur de références différent de zéro ne sera normalement pas libérée. Par conséquent, ceci est une mauvaise idée :

```
{
 my $a;
 $a = \$a;
}
```

Alors même que la variable `$a` devrait disparaître, elle ne le peut pas. Lorsque vous construisez des structures de données récursives, vous devrez briser vous-même explicitement l'auto-référence si vous ne voulez pas de fuite de mémoire. Par exemple, voici un noeud auto-référent comme ceux qu'on pourrait utiliser dans une structure d'arbre sophistiquée :

```
sub new_node {
 my $self = shift;
 my $class = ref($self) || $self;
 my $node = {};
 $node->{LEFT} = $node->{RIGHT} = $node;
 $node->{DATA} = [@_];
 return bless $node => $class;
}
```

Si vous créez de tels noeuds, ils ne disparaîtront pas (actuellement) à moins que vous ne brisiez leur auto-référence vous-même (en d'autres termes, cela ne doit pas être considéré comme une caractéristique et vous ne devriez pas compter là-dessus).

Ou presque.

Lorsqu'un thread de l'interpréteur se termine finalement (habituellement au moment où votre programme se termine), une libération de la mémoire plutôt coûteuse, mais complète par marquage et nettoyage est effectuée, tout ce qui a été alloué par ce thread est détruit. C'est essentiel pour pouvoir supporter Perl comme un langage embarqué et multithread. Par exemple, ce programme montre le ramassage des miettes en deux phases de Perl :

```
#!/usr/bin/perl
package Subtle;

sub new {
 my $test;
 $test = \$test;
 warn "CREATING " . \$test;
 return bless \$test;
}

sub DESTROY {
 my $self = shift;
 warn "DESTROYING $self";
}

package main;
```



```
warn "starting program";
{
 my $a = Subtle->new;
 my $b = Subtle->new;
 $$a = 0; # break selfref
 warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

Exécuté en tant que `/tmp/test`, la sortie suivante est produite :

```
starting program at /tmp/test line 18.
CREATING SCALAR(0x8e5b8) at /tmp/test line 7.
CREATING SCALAR(0x8e57c) at /tmp/test line 7.
leaving block at /tmp/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /tmp/test line 13.
just exited block at /tmp/test line 26.
time to die... at /tmp/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Avez-vous remarqué le "global destruction" ? C'est le ramasse-miettes du thread en train d'atteindre l'inaccessible.

Les objets sont toujours détruits, même lorsque les références normales ne le sont pas. Les objets sont supprimés lors d'une passe distincte avant les références ordinaires juste pour éviter aux destructeurs d'objets d'utiliser des références ayant déjà été elles-mêmes détruites. Les simples références ne sont supprimées que si le niveau de destruction est supérieur à 0. Vous pouvez tester les plus hauts niveaux de destruction globale en fixant la variable d'environnement `PERL_DESTRUCT_LEVEL`, si `-DDEBUGGING` a été utilisée lors de la compilation de perl. Voir `PERL_DESTRUCT_LEVEL` in *perlhack* pour plus d'information.

Une stratégie plus complète de ramassage des miettes sera implémentée un jour.

En attendant, la meilleure solution est de créer une classe de conteneur non-récuratif détenant un pointeur vers la structure de données auto-référentielle. Puis, de définir une méthode `DESTROY` pour la classe de conteneurs qui brise manuellement les circularités dans la structure auto-référentielle.

## 42.2 VOIR AUSSI

Des tutoriels plus doux et plus gentils sur la programmation orientée objet en Perl se trouvent dans *perltoot*, *perlboot* et *perltooc*. Vous devriez aussi jeter un oeil sur *perlbob* pour d'autres petits trucs concernant les objets, les pièges et les astuces, ainsi que sur *perlmodlib* pour des guides de style sur la construction de modules et de classes.

## 42.3 TRADUCTION

### 42.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 42.3.2 Traducteur

Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit (Paul.Gaborit @ enstimac.fr).

### 42.3.3 Relecture

Philippe de Visme <[philippe@devisme.com](mailto:philippe@devisme.com)>

# Chapitre 43

## perltie

Comment cacher un objet d'une classe derrière une simple variable

### 43.1 SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST
```

```
$object = tied VARIABLE
```

```
untie VARIABLE
```

### 43.2 DESCRIPTION

Avant la version 5.0 de Perl, un programmeur pouvait utiliser la fonction `dbmopen()` pour connecter magiquement une table de hachage `%HASH` de son programme à une base de données sur disque au format standard Unix `dbm(3x)`. En revanche, son Perl était construit avec l'une ou l'autre des implémentations de la bibliothèque `dbm` mais pas les deux et il n'était pas possible d'étendre ce mécanisme à d'autres packages ou à d'autres types de variables.

C'est maintenant possible.

La fonction `tie()` relie une variable à une classe (ou un package) qui fournit l'implémentation des méthodes permettant d'accéder à cette variable. Une fois ce lien magique établi, l'accès à cette variable déclenche automatiquement l'appel aux méthodes de la classe choisie. La complexité de la classe est cachée derrière des appels magiques de méthodes. Les noms de ces méthodes sont entièrement en MAJUSCULES. C'est une convention pour signifier que Perl peut appeler ces méthodes implicitement plutôt qu'explicitement – exactement comme les fonctions `BEGIN()` et `END()`.

Dans l'appel de `tie()`, `VARIABLE` est le nom de la variable à relier. `CLASSNAME` est le nom de la classe qui implémente les objets du bon type. Tous les autres arguments dans `LIST` sont passés au constructeur approprié pour cette classe – à savoir `TIESCALAR()`, `TIEARRAY()`, `TIEHASH()`, ou `TIEHANDLE()`. (Ce sont par exemple les arguments à passer à la fonction C `dbmopen()`.) L'objet retourné par la méthode "new" est aussi retourné par la fonction `tie()` ce qui est pratique pour accéder à d'autres méthodes de la classe `CLASSNAME`. (Vous n'avez pas besoin de retourner une référence vers un vrai "type" (e.g. `HASH` ou `CLASSNAME`) tant que c'est un objet correctement béni – par la fonction `bless()`.) Vous pouvez aussi retrouver une référence vers l'objet sous-jacent en utilisant la fonction `tied()`.

À la différence de `dbmopen()`, la fonction `tie()` ne fera pas pour vous un `use` ou un `require` d'un module – vous devez le faire vous-même explicitement.

#### 43.2.1 Les scalaires liés (par `tie()`)

Une classe qui implémente un scalaire lié devrait définir les méthodes suivantes : `TIESCALAR`, `FETCH`, `STORE` et éventuellement `UNTIE` et `DESTROY`.

Jetons un oeil à chacune d'elle en utilisant comme exemple une classe liée (par `tie()`) qui permet à l'utilisateur de faire :

```
tie $his_speed, 'Nice', getppid();
tie $my_speed, 'Nice', $$;
```

À partir de maintenant, à chaque accès à l'une de ces variables, la priorité courante du système est retrouvée et retournée. Si ces variables sont modifiées alors la priorité du process change.

Nous utilisons la classe BSD::Resource de Jarkko Hietaniemi <jhi@iki.fi> (non fournie) pour accéder aux constantes systèmes PRIO\_PROCESS, PRIO\_MIN, et PRIO\_MAX ainsi qu'aux appels systèmes getpriority() et setpriority(). Voici le préambule de la classe :

```
package Nice;
use Carp;
use BSD::Resource;
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

### TIESCALAR classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par bless()) vers un nouveau scalaire (probablement anonyme) qu'il a créé. Par exemple :

```
sub TIESCALAR {
 my $class = shift;
 my $pid = shift || $$; # 0 means me

 if ($pid !~ /\d+$/) {
 carp "Nice::Tie::Scalar got non-numeric pid $pid" if $^W;
 return undef;
 }

 unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
 carp "Nice::Tie::Scalar got bad pid $pid: $!" if $^W;
 return undef;
 }

 return bless \$pid, $class;
}
```

Cette classe liée a choisi de retourner une erreur plutôt que de lancer une exception si son constructeur échoue. Bien que ce soit la manière dont dbmopen() fonctionne, d'autres classes peuvent très bien être moins tolérantes. La classe regarde tout de même la variable \$^W pour savoir si elle doit émettre un peu de bruit ou non.

### FETCH this

Cette méthode se déclenche chaque fois qu'on accède (en lecture) à une variable liée. Elle ne prend aucun argument mis à part une référence qui est l'objet qui représente le scalaire à utiliser. Puisque dans notre cas nous utilisons une simple référence SCALAIRE comme objet du scalaire lié, un appel à \$\$self permet à la méthode d'obtenir la valeur réelle stockée. Dans notre exemple ci-dessous, cette valeur réelle est l'ID du process que nous avons lié à notre variable.

```
sub FETCH {
 my $self = shift;
 confess "wrong type" unless ref $self;
 croak "usage error" if @_;
 my $nicety;
 local($!) = 0;
 $nicety = getpriority(PRIO_PROCESS, $$self);
 if ($!) { croak "getpriority failed: $!" }
 return $nicety;
}
```

Cette fois, nous avons décidé d'exploser (en générant une exception) si l'obtention de la priorité échoue – il n'y a aucun autre moyen pour retourner une erreur et c'est probablement la meilleure chose à faire.

### STORE this, value

Cette méthode est appelée à chaque fois que la variable liée est modifiée (affectée). Derrière sa propre référence, elle attend un (et un seul) argument : la nouvelle valeur que l'utilisateur essaye d'affecter. Ne vous préoccupez pas de la valeur retournée par STORE : le fait qu'une affectation retourne la valeur qui vient d'être affectée est implémenté en passant par FETCH.

```

sub STORE {
 my $self = shift;
 confess "wrong type" unless ref $self;
 my $new_nicety = shift;
 croak "usage error" if @_;

 if ($new_nicety < PRIO_MIN) {
 carp sprintf
 "WARNING: priority %d less than minimum system priority %d",
 $new_nicety, PRIO_MIN if $^W;
 $new_nicety = PRIO_MIN;
 }

 if ($new_nicety > PRIO_MAX) {
 carp sprintf
 "WARNING: priority %d greater than maximum system priority %d",
 $new_nicety, PRIO_MAX if $^W;
 $new_nicety = PRIO_MAX;
 }

 unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
 confess "setpriority failed: $!";
 }
}

```

**UNTIE this**

Cette méthode sera appelée lorsqu'un untie aura lieu. Cela peut être pratique pour la classe a besoin de savoir qu'on ne l'appellera plus (sauf via DESTROY évidemment). Voir Les pièges du untie ci-dessous pour plus de détails.

**DESTROY this**

Cette méthode est appelée lorsque la variable liée doit être détruite. Comme pour les autres classes d'objets, une telle méthode est rarement nécessaire parce que Perl désalloue automatiquement pour vous la mémoire associé à des objets moribonds – ce n'est pas le cas de C++. Nous utilisons donc une méthode DESTROY ici uniquement pour déboguer.

```

sub DESTROY {
 my $self = shift;
 confess "wrong type" unless ref $self;
 carp "[Nice::DESTROY pid $$self]" if $Nice::DEBUG;
}

```

C'est tout ce qu'il y a à voir. C'est même un peu plus parce que nous avons fait des jolies petites choses dans un souci de complétude, robustesse et, plus généralement, d'esthétique. Des classes plus simples liant un scalaire sont certainement faisables.

**43.2.2 Les tableaux liés (par tie())**

Une classe qui implémente un tableau lié ordinaire devrait définir les méthodes suivantes : TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE et éventuellement UNTIE et DESTROY.

Les méthodes FETCHSIZE et STORESIZE sont nécessaires pour pouvoir fournir \$#array et autres fonctionnalités équivalentes comme scalar(@array).

Les méthodes POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE et EXISTS sont indispensables si l'opérateur Perl ayant le même nom (mais en minuscules) doit être appliqué aux tableaux liés. La classe **Tie::Array** peut être utilisée comme classe de base pour implémenter ces méthodes à partir des cinq méthodes initiales de base. Les implémentations par défaut de DELETE et de EXISTS dans **Tie::Array** appellent simplement croak.

De plus, la méthode EXTEND sera appelée quand perl devra pré-étendre l'allocation du tableau.

Pour illustrer cela, nous allons implémenter un tableau dont la longueur des éléments est constante et fixée lors de la création du tableau. Si on tente de créer un élément plus grand que cette limite, cela produira une exception. Par exemple :

```

use FixedElem_Array;
tie @array, 'FixedElem_Array', 3;
$array[0] = 'cat'; # ok.
$array[1] = 'dogs'; # exception, length('dogs') > 3.

```

Le code en préambule de cette classe est le suivant :

```
package FixedElem_Array;
use Carp;
use strict;
```

### **TIEARRAY classname, LIST**

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par `bless()`) à travers laquelle on pourra accéder au nouveau tableau (probablement une référence à un TABLEAU anonyme)

Dans notre exemple, juste pour montrer qu'il n'est pas *VRAIMENT* nécessaire de retourner une référence vers un TABLEAU, nous avons choisi une référence à une HASHTABLE pour représenter notre objet. Cette HASHTABLE s'élabore exactement comme un type d'enregistrement générique : le champ `{ELEM_SIZE}` stockera la taille maximale autorisée et le champ `{ARRAY}` contiendra la référence vers le vrai TABLEAU. Si quelqu'un en dehors de la classe tente de déréférencer l'objet retourné (croyant sans doute avoir à faire à une référence vers un TABLEAU), cela ne fonctionnera pas. Tout cela pour vous montrer que vous devriez respecter le vie privée des objets.

```
sub TIEARRAY {
 my $class = shift;
 my $elemsize = shift;
 if (@_ || $elemsize =~ /\D/) {
 croak "usage: tie ARRAY, '" . __PACKAGE__ . "', elem_size";
 }
 return bless {
 ELEM_SIZE => $elemsize,
 ARRAY => [],
 }, $class;
}
```

### **FETCH this, indice**

Cette méthode est appelée à chaque fois qu'on accède (en lecture) à un élément individuel d'un tableau lié. Elle prend un argument en plus de sa propre référence : l'indice de la valeur à laquelle on veut accéder.

```
sub FETCH {
 my $self = shift;
 my $indice = shift;
 return $self->{ARRAY}->[$indice];
}
```

Si un indice négatif est utilisé par lire dans un tableau, l'indice est converti de manière interne en une valeur positive en appelant d'abord `FETCHSIZE` avant de passer l'indice résultant à `FETCH`. Vous pouvez désactiver cette fonctionnalité en affectant une valeur vraie à la variable `$NEGATIVE_INDICES` de la classe du tableau lié.

Comme vous avez pu le remarquer le nom de la méthode `FETCH` (et al.) est le même pour tous les accès bien que les constructeurs ont des noms différents (`TIESCALAR` vs `TIEARRAY`). En théorie, une même classe peut servir pour différents types d'objets liés. En pratique, cela devient rapidement encombrant et il est beaucoup plus simple de n'avoir qu'un seul type d'objets liés par classe.

### **STORE this, indice, value**

Cette méthode est appelée à chaque fois qu'un élément d'un tableau lié est modifié (affecté). Elle prend deux arguments en plus de sa propre référence : l'indice de l'élément où nous voulons stocker quelque chose et la valeur que nous voulons stocker.

Dans notre exemple, `undef` est en fait un nombre d'espaces égale à `$self->{ELEM_SIZE}`. Nous avons donc un peu de travail à faire ici...

```
sub STORE {
 my $self = shift;
 my($indice, $value) = @_;
 if (length $value > $self->{ELEM_SIZE}) {
 croak "la longueur de $value est superieure a $self->{ELEM_SIZE}";
 }
 # remplir les trous
 $self->EXTEND($indice) if $indice > $self->FETCHSIZE();
 # aligner à droite pour les éléments les plus petits
 $self->{ARRAY}->[$indice] = sprintf "%$self->{ELEM_SIZE}s", $value;
}
```

Les indices négatifs sont traités de la même manière qu'avec FETCH.

### FETCHSIZE this FETCHSIZE

Retourne le nombre total d'éléments présents dans le tableau associé à l'objet *this*. (Similaire à `scalar(@array)`.)  
Par exemple :

```
sub FETCHSIZE {
 my $self = shift;
 return scalar @{$self->{ARRAY}};
}
```

### STORESIZE this, count

Fixe à *count* le nombre total d'éléments stockés dans le tableau associé à l'objet *this*. Si cela aggrandit le tableau, la valeur undef proposée par la classe devrait être retournée pour les nouveaux emplacements créés. Si le tableau diminue alors les éléments en trop sont supprimés.

Dans notre exemple, la valeur 'undef' est en fait un suite de `$self->{ELEM_SIZE}` espaces. Ce qui donne :

```
sub STORESIZE {
 my $self = shift;
 my $count = shift;
 if ($count > $self->FETCHSIZE()) {
 foreach ($count - $self->FETCHSIZE() .. $count) {
 $self->STORE($_, ' ');
 }
 } elsif ($count < $self->FETCHSIZE()) {
 foreach (0 .. $self->FETCHSIZE() - $count - 2) {
 $self->POP();
 }
 }
}
```

### EXTEND this, count

C'est un appel informatif permettant d'indiquer que le nombre d'éléments du tableau risque d'atteindre *count*. Ça peut servir pour optimiser l'utilisation de la mémoire. Cette méthode n'est pas obligée de faire quelque chose.

Dans notre exemple, nous voulons être sûr qu'il n'y a pas d'éléments vide (ou undef). Donc EXTEND fait appel à STORESIZE pour remplir les éléments comme il faut :

```
sub EXTEND {
 my $self = shift;
 my $count = shift;
 $self->STORESIZE($count);
}
```

### EXISTS this, key

Permet de vérifier si l'élément dont l'indice est *key* existe dans le tableau lié à *this*.

Dans notre exemple, nous décidons qu'un élément constitué uniquement de `$self->{ELEM_SIZE}` espaces n'existe pas :

```
sub EXISTS {
 my $self = shift;
 my $index = shift;
 return 0 if ! defined $self->{ARRAY}->[$index] ||
 $self->{ARRAY}->[$index] eq ' ' x $self->{ELEM_SIZE};
 return 1;
}
```

### DELETE this, key

Efface l'élément du tableau lié à *this* dont l'indice est *key*.

Dans notre exemple, un élément effacé contient `$self->{ELEM_SIZE}` espaces :

```
sub DELETE {
 my $self = shift;
 my $index = shift;
 return $self->STORE($index, ' ');
}
```

**CLEAR this**

Efface (supprime, détruit, ...) tous les éléments stockés dans le tableau lié à l'objet *this*. Par exemple :

```
sub CLEAR {
 my $self = shift;
 return $self->{ARRAY} = [];
}
```

**PUSH this, LIST**

Ajoute tous les éléments de *LIST* au tableau. Par exemple :

```
sub PUSH {
 my $self = shift;
 my @list = @_;
 my $last = $self->FETCHSIZE();
 $self->STORE($last + $_, $list[$_]) foreach 0 .. $#list;
 return $self->FETCHSIZE();
}
```

**POP this**

Supprime et retourne le dernier élément du tableau. Par exemple :

```
sub POP {
 my $self = shift;
 return pop @{$self->{ARRAY}};
}
```

**SHIFT this**

Supprime et retourne le premier élément du tableau (en décalant les éléments suivants). Par exemple :

```
sub SHIFT {
 my $self = shift;
 return shift @{$self->{ARRAY}};
}
```

**UNSHIFT this, LIST**

Insère tous les éléments de *LIST* au début du tableau en décalant les éléments existants pour faire de la place. Par exemple :

```
sub UNSHIFT {
 my $self = shift;
 my @list = @_;
 my $size = scalar(@list);
 # on fait de la place pour la liste
 @{$self->{ARRAY}}[$size .. @{$self->{ARRAY}} + $size]
 = @{$self->{ARRAY}};
 $self->STORE($_, $list[$_]) foreach 0 .. $#list;
}
```

**SPLICE this, offset, length, LIST**

Réalise l'équivalent d'un splice sur le tableau.

*offset* est optionnel et vaut zéro par défaut. Les valeurs négatives sont comptés à partir de la fin du tableau.

*length* est optionnel et, par défaut, va jusqu'à la fin du tableau.

*LIST* peut être vide.

Retourne la liste des éléments originaux débutant à *offset* et de longueur *length*.

Dans notre exemple, on utilise un petit raccourci si il y a une *LIST* :

```
sub SPLICE {
 my $self = shift;
 my $offset = shift || 0;
 my $length = shift || $self->FETCHSIZE() - $offset;
 my @list = ();
 if (@_) {
 tie @list, __PACKAGE__, $self->{ELEMENTS};
 }
}
```

```

 @list = @_;
 }
 return splice @{$self->{ARRAY}}, $offset, $length, @list;
}

```

**UNTIE this**

Sera appelée lorsqu'un untie se produira. (Voir Les pièges du untie ci-dessous.)

**DESTROY this**

Cette méthode est appelée lorsque qu'une variable liée doit être détruite. Comme dans le cas des scalaires, cela est rarement nécessaire avec un langage qui a son propre ramasse-miettes. Donc, cette fois nous la laisserons de côté.

**43.2.3 Les tables de hachage liées (par tie())**

Les tables de hachage ont été le premier type de données en Perl à pouvoir être lié (voir dbmopen()). Une classe qui implémente une table de hachage liée devrait définir les méthodes suivantes : TIEHASH est le constructeur ; FETCH et STORE accèdent aux paires clés/valeurs. EXISTS indique si une clé est présente dans la table et DELETE en supprime une. CLEAR vide la table en supprimant toutes les paires clé/valeur. FIRSTKEY et NEXTKEY implémentent les fonctions keys() et each() pour parcourir l'ensemble des clés. SCALAR est appelé lorsque la table de hachage liée est évaluée dans un contexte scalaire. UNTIE est appelé lorsqu'un untie a lieu, et DESTROY est appelée lorsque la variable liée est détruite.

Si cela vous semble beaucoup, vous pouvez hériter du module standard Tie::StdHash pour la plupart de vos méthodes et ne redéfinir que celles qui vous intéressent. Voir *Tie::Hash* pour plus de détails.

Souvenez-vous que Perl distingue le cas où une clé n'existe pas dans la table de celui où la clé existe mais correspond à une valeur undef. Les deux possibilités peuvent être testées via les fonctions exists() et defined().

Voici l'exemple relativement intéressant d'une classe liant une table de hachage : cela vous fournit une table de hachage représentant tous les fichiers .\* d'un utilisateur. Vous donnez comme index dans la table le nom du fichier (le point en moins) et vous récupérez le contenu de ce fichier. Par exemple :

```

use DotFiles;
tie %dot, 'DotFiles';
if ($dot{profile} =~ /MANPATH/ ||
 $dot{login} =~ /MANPATH/ ||
 $dot{cshrc} =~ /MANPATH/)
{
 print "you seem to set your MANPATH\n";
}

```

Ou une autre utilisation possible de notre classe liée :

```

tie %him, 'DotFiles', 'daemon';
foreach $f (keys %him) {
 printf "daemon dot file %s is size %d\n",
 $f, length $him{$f};
}

```

Dans notre exemple de table de hachage liée DotFiles, nous utilisons une table de hachage normale pour stocker dans l'objet plusieurs champs importants dont le champ {LIST} qui apparaît à l'utilisateur comme le contenu réel de la table.

**USER**

l'utilisateur pour lequel l'objet représente les fichiers .\*

**HOME**

l'endroit où se trouve ces fichiers

**CLOBBER**

si nous pouvons essayer de modifier ou de supprimer ces fichiers.

**LIST**

la table de hachage des noms des fichiers .\* et de leur contenu

Voici le début de *Dotfiles.pm* :



```

package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . '()' }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }

```

Dans notre exemple, nous voulons avoir la possibilité d'émettre des informations de debug pour aider au développement. Nous fournissons aussi une fonction pratique pour aider à l'affichage des messages d'avertissements ; whowasi() retourne le nom de la fonction qui l'a appelé.

Voici les méthodes pour la table de hachage DotFiles.

### TIEHASH classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par bless()) au travers de laquelle on peut accéder au nouvel objet (probablement mais pas nécessairement une table de hachage anonyme).

Voici le constructeur :

```

sub TIEHASH {
 my $self = shift;
 my $user = shift || $>;
 my $dotdir = shift || '';
 croak "usage: @{{&whowasi}} [USER [DOTDIR]]" if @_;
 $user = getpwuid($user) if $user =~ /^d+$/;
 my $dir = (getpwnam($user))[7]
 || croak "@{{&whowasi}}: no user $user";
 $dir .= "/$dotdir" if $dotdir;

 my $node = {
 USER => $user,
 HOME => $dir,
 LIST => {},
 CLOBBER => 0,
 };

 opendir(DIR, $dir)
 || croak "@{{&whowasi}}: can't opendir $dir: $!";
 foreach $dot (grep /^\.\/ && -f "$dir/$_", readdir(DIR)) {
 $dot =~ s/^\./;
 $node->{LIST}{$dot} = undef;
 }
 closedir DIR;
 return bless $node, $self;
}

```

Il n'est pas inutile de préciser que, si vous voulez tester un fichier dont le nom est produit par readdir, vous devez la précéder du nom du répertoire en question. Sinon, puisque nous ne faisons pas de chdir() ici, il ne testera sans doute pas le bon fichier.

### FETCH this, key

Cette méthode est appelée à chaque fois qu'on accède (en lecture) à un élément d'une table de hachage liée. Elle prend un argument en plus de sa propre référence : la clé d'accès à la valeur que l'on cherche à lire.

Voici le code FETCH pour notre exemple DotFiles.

```

sub FETCH {
 carp &whowasi if $DEBUG;
 my $self = shift;
 my $dot = shift;
 my $dir = $self->{HOME};
 my $file = "$dir/$dot";

 unless (exists $self->{LIST}->{$dot} || -f $file) {
 carp "@{{&whowasi}}: no $dot file" if $DEBUG;
 return undef;
 }
}

```

```

 if (defined $self->{LIST}->{$dot}) {
 return $self->{LIST}->{$dot};
 } else {
 return $self->{LIST}->{$dot} = 'cat $dir/.$dot';
 }
}

```

C'est facile à écrire en lui faisant appeler la commande Unix `cat(1)` mais ce serait probablement plus portable d'ouvrir le fichier manuellement (et peut-être plus efficace). Bien sûr, puisque les fichiers `.*` sont un concept Unix, nous ne sommes pas concernés.

### STORE this, key, value

Cette méthode est appelée à chaque accès (en écriture) à un élément d'une table de hachage liée. Elle prend deux arguments en plus de sa propre référence : la clé qui nous servira pour stocker quelque chose et la valeur que vous lui associez.

Dans notre exemple `DotFiles`, nous n'autorisons la réécriture du fichier que dans le cas où la méthode `clobber()` a été appelée à partir de la référence objet retournée par `tie()`.

```

sub STORE {
 carp &whowasi if $DEBUG;
 my $self = shift;
 my $dot = shift;
 my $value = shift;
 my $file = $self->{HOME} . "/.$dot";
 my $user = $self->{USER};

 croak "@{&whowasi}: $file not clobberable"
 unless $self->{CLOBBER};

 open(F, "> $file") || croak "can't open $file: $!";
 print F $value;
 close(F);
}

```

Si quelqu'un veut écraser quelque chose, il doit dire :

```

$obj = tie %daemon_dots, 'daemon';
$obj->clobber(1);
$daemon_dots{signature} = "A true daemon\n";

```

Un autre moyen de mettre la main sur une référence à l'objet sous-jacent est d'utiliser la fonction `tied()`. Il serait donc aussi possible de dire :

```

tie %daemon_dots, 'daemon';
tied(%daemon_dots)->clobber(1);

```

La méthode `clobber` est très simple :

```

sub clobber {
 my $self = shift;
 $self->{CLOBBER} = @_ ? shift : 1;
}

```

### DELETE this, key

Cette méthode est appelée lorsqu'on supprime un élément d'une table de hachage en utilisant par exemple la fonction `delete()`. Encore une fois, nous vérifions que nous voulons réellement écraser les fichiers.

```

sub DELETE {
 carp &whowasi if $DEBUG;

 my $self = shift;
 my $dot = shift;
 my $file = $self->{HOME} . "/.$dot";
 croak "@{&whowasi}: won't remove file $file"
 unless $self->{CLOBBER};
 delete $self->{LIST}->{$dot};
 my $success = unlink($file);
}

```

```

 carp "@{&whowasi}: can't unlink $file: $" unless $success;
 $success;
}

```

La valeur retournée par DELETE deviendra la valeur retournée par l'appel à delete(). Si vous voulez simuler le comportement normal de delete(), vous devriez retourner ce qu'aurait retourné FETCH pour cette clé. Dans notre exemple, nous avons préféré retourner une valeur qui indique à l'appelant si le fichier a été correctement effacé.

#### CLEAR this

Cette méthode est appelée lorsque l'ensemble de la table de hachage est effacée, habituellement en lui affectant la liste vide.

Dans notre exemple, cela devrait supprimer tous les fichiers .\* de l'utilisateur ! C'est une chose tellement dangereuse que nous exigeons un positionnement de CLOBBER à une valeur supérieure à 1 pour l'autoriser.

```

sub CLEAR {
 carp &whowasi if $DEBUG;
 my $self = shift;
 croak "@{&whowasi}: won't remove all dot files for $self->{USER}"
 unless $self->{CLOBBER} > 1;
 my $dot;
 foreach $dot (keys %{$self->{LIST}}) {
 $self->DELETE($dot);
 }
}

```

#### EXISTS this, key

Cette méthode est appelée lorsque l'utilisateur appelle la fonction exists() sur une table pour une clé particulière. Dans notre exemple, nous cherchons la clé dans la table de hachage {LIST} :

```

sub EXISTS {
 carp &whowasi if $DEBUG;
 my $self = shift;
 my $dot = shift;
 return exists $self->{LIST}->{$dot};
}

```

#### FIRSTKEY this

Cette méthode est appelée lorsque l'utilisateur commence une itération à travers la table de hachage via un appel à keys() ou each().

```

sub FIRSTKEY {
 carp &whowasi if $DEBUG;
 my $self = shift;
 my $a = keys %{$self->{LIST}}; # reset each() iterator
 each %{$self->{LIST}}
}

```

#### NEXTKEY this, lastkey

Cette méthode est appelée lors d'une itération via keys() ou each(). Son second argument est la dernière clé à laquelle on a accédé. C'est pratique si vous voulez faire attention à l'ordre, si vous appelez l'itérateur pour plusieurs séquences à la fois ou si vous ne stockez pas vos données dans une table de hachage réelle.

Dans notre exemple, nous utilisons une vraie table de hachage. Nous pouvons donc faire simplement en accédant tout de même au champ LIST de manière indirecte.

```

sub NEXTKEY {
 carp &whowasi if $DEBUG;
 my $self = shift;
 return each %{$self->{LIST}}
}

```

#### SCALAR this

Cette méthode est appelée lorsqu'on évalue la table de hachage dans un contexte scalaire. Afin d'adopter un comportement similaire à celui des tables de hachage normales, cette méthode devrait retourner une valeur fausse lorsque la table de hachage liée peut être considérée comme étant vide. Si cette méthode n'existe pas, perl tentera d'adopter

un comportement compatible et retournera vrai si la table de hachage est en cours d'itération. Si ce n'est pas le cas, FIRSTKEY sera appelé et le résultat sera une valeur fausse si FIRSTKEY retourne une liste vide, et une valeur vraie sinon.

Par contre, **ne** vous reposez **pas** aveuglément sur ce mécanisme en espérant que perl s'en sortira toujours. Par exemple, perl continuera à retourner une valeur vraie si vous videz la table de hachage par une suite d'appels à DELETE. Vous devez donc fournir vous-même une bonne méthode SCALAR si vous souhaitez un comportement correct dans un contexte scalaire.

Dans notre exemple, il nous suffit d'appeler scalar en l'appliquant à la table de hachage sous-jacente référencée par \$self->{LIST} :

```
sub SCALAR {
 carp &whowasi if $DEBUG;
 my $self = shift;
 return scalar %{ $self->{LIST} }
}
```

### UNTIE this

Sera appelée lorsqu'un untie se produira. (Voir Les pièges du untie ci-dessous.)

### DESTROY this

Cette méthode est appelée lorsque une table de hachage liée va disparaître. Vous n'en avez pas réellement besoin sauf pour déboguer ou pour nettoyer quelques données auxiliaires. Voici une fonction très simple :

```
sub DESTROY {
 carp &whowasi if $DEBUG;
}
```

Notez que des fonctions telles que keys() et values() peuvent retourner des listes énormes lorsqu'elles sont utilisées sur de gros objets comme des fichiers DBM. Vous devriez plutôt utiliser la fonction each() pour les parcourir. Exemple :

```
print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
 print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

## 43.2.4 Les descripteurs de fichiers (filehandles) liés (par tie())

Pour l'instant, ce n'est que partiellement implémenté.

Une classe qui implémente un filehandle lié devrait définir les méthodes suivantes : TIEHANDLE, au moins une méthode parmi PRINT, PRINTF, WRITE, READLINE, GETC, READ, et éventuellement CLOSE, UNTIE et DESTROY. La classe peut aussi fournir : BINMODE, OPEN, EOF, FILENO, SEEK et TELL (si les opérateurs Perl correspondants sont utilisés sur le descripteur).

Lorsque STDERR est lié, c'est sa méthode PRINT qui sera appelée à chaque émission d'un message d'avertissement ou d'erreur. Cette fonctionnalité est temporairement désactivée durant cet appel ce qui signifie que vous pouvez faire appel à warn() à l'intérieur de PRINT sans risquer une série d'appels récursifs. Notez aussi que, comme les gestionnaires de signaux \_\_WARN\_\_ et \_\_DIE\_\_, la méthode PRINT de STDERR peut être appelée pour indiquer des erreurs de compilations. Par conséquent, toutes les remarques faites dans %SIG in *perlvar* s'appliquent.

Tout cela est particulièrement pratique lorsque perl est utilisé à l'intérieur d'un autre programme dans lequel STDOUT et STDERR doivent être redirigés d'une manière un peu spéciale. Voir nvi et le module Apache par exemple.

Dans l'exemple suivant, nous essayons de créer un filehandle hurleur (*shouting* en anglais).

```
package Shout;
```

### TIEHANDLE classname, LIST

C'est le constructeur de la classe. Cela signifie qu'il est supposé retourner une référence bénie (par bless()). Cette référence peut être utilisée pour stocker des informations internes.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

**WRITE this, LIST**

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `syswrite`.

```
sub WRITE {
 $r = shift;
 my($buf,$len,$offset) = @_;
 print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

**PRINT this, LIST**

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `print()`. En plus de sa propre référence, elle attend une liste à passer à la fonction `print`.

```
sub PRINT { $r = shift; $$r++; print join(,,map(uc($_),@_)),$\ }
```

**PRINTF this, LIST**

Cette méthode est appelée lorsqu'on écrit sur le filehandle via la fonction `printf()`. En plus de sa propre référence, elle attend un format et une liste à passer à la fonction `printf`.

```
sub PRINTF {
 shift;
 my $fmt = shift;
 print sprintf($fmt, @_);
}
```

**READ this, LIST**

Cette méthode est appelée lorsque le filehandle est lu via les fonctions `read()` ou `sysread()`.

```
sub READ {
 my $self = shift;
 my $bufref = $_[0];
 my(undef,$len,$offset) = @_;
 print "READ called, \$buf=$bufref, \$len=$len, \$offset=$offset";
 # add to $$bufref, set $len to number of characters read
 $len;
}
```

**READLINE this**

Cette méthode est appelée lorsque le filehandle est lu via `<HANDLE>`. Elle devrait retourner `undef` lorsque il n'y a plus de données.

```
sub READLINE { $r = shift; "PRINT called $$r times\n"; }
```

**GETC this**

Cette méthode est appelée lorsque la fonction `getc` est appelée.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

**CLOSE this**

Cette méthode est appelée lorsque le filehandle est fermé via la fonction `close`.

```
sub CLOSE { print "CLOSE called.\n" }
```

**UNTIE this**

Comme pour les autres types de liaisons, cette méthode sera appelée lorsqu'un appel à `untie` aura lieu. Il peut être intéressant de faire un "auto CLOSE" à ce moment-là. Voir [Les pièges de untie](#) ci-dessous.

**DESTROY this**

Comme pour les autres types de variables liées, cette méthode sera appelée lorsque le filehandle lié va être détruit. C'est pratique pour déboguer et éventuellement nettoyer.

```
sub DESTROY { print "</shout>\n" }
```

Voici comment utiliser notre petit exemple :

```
tie(*F00,'Shout');
print F00 "hello\n";
$a = 4; $b = 6;
print F00 $a, " plus ", $b, " equals ", $a + $b, "\n";
print <F00>;
```

### 43.2.5 Les pièges du untie

Si vous projetez d'utiliser l'objet retourné soit par tie() soit par tied() et si la classe liée définit un destructeur, il y a un piège subtile que vous *devez* connaître.

Comme base, considérons cette exemple d'utilisation de tie; la seule chose qu'il fait est de garder une trace de toutes les valeurs affectées à un scalaire.

```
package Remember;

use strict;
use IO::File;

sub TIESCALAR {
 my $class = shift;
 my $filename = shift;
 my $handle = IO::File->new("> $filename")
 or die "Cannot open $filename: !\n";

 print $handle "The Start\n";
 bless {FH => $handle, Value => 0}, $class;
}

sub FETCH {
 my $self = shift;
 return $self->{Value};
}

sub STORE {
 my $self = shift;
 my $value = shift;
 my $handle = $self->{FH};
 print $handle "$value\n";
 $self->{Value} = $value;
}

sub DESTROY {
 my $self = shift;
 my $handle = $self->{FH};
 print $handle "The End\n";
 close $handle;
}

1;
```

Voici un exemple d'utilisation :

```
use strict;
use Remember;

my $fred;
tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

Et voici le résultat de l'exécution :

```
The Start
1
4
5
The End
```

Jusqu'à maintenant, tout va bien. Pour ceux qui ne l'auraient pas remarqué, l'objet lié n'a pas encore été utilisé. Ajoutons donc, à la classe Remember, une méthode supplémentaire permettant de mettre des commentaires dans le fichier – disons quelque chose comme :

```
sub comment {
 my $self = shift;
 my $text = shift;
 my $handle = $self->{FH};
 print $handle $text, "\n";
}
```

Voici maintenant l'exemple précédent modifié pour utiliser la méthode comment (qui nécessite l'objet lié) :

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

Lorsque ce code s'exécute, il ne produit rien. Voici pourquoi...

Lorsqu'une variable est liée, elle est associée avec l'objet qui est retourné par la fonction TIESCALAR, TIEARRAY, ou TIEHASH. Cette objet n'a normalement qu'une seule référence, à savoir, la référence implicite prise par la variable liée. Lorsque untie() est appelé, cette référence est supprimée. Et donc, comme dans notre premier exemple ci-dessus, le destructeur (DESTROY) de l'objet est appelé ce qui est normal pour un objet qui n'a plus de référence valide; et donc le fichier est fermé.

Dans notre second exemple, par contre, nous avons stocké dans \$x une autre référence à l'objet lié. Si bien que lorsque untie() est appelé, il reste encore une référence valide sur l'objet, donc le destructeur n'est pas appelé à ce moment et donc le fichier n'est pas fermé. Le buffer du fichier n'étant pas vidé, ceci explique qu'il n'y ait pas de sortie.

Bon, maintenant que nous connaissons le problème, que pouvons-nous faire pour l'éviter? Avant l'introduction de la méthode optionnelle UNTIE, il n'y avait que la bonne vieille option -w. Celle-ci vous signale tout appel à untie() pour lequel l'objet lié possède encore des références valides. Si le second script ci-dessus est utilisé avec l'option -w, Perl affiche le message d'avertissement suivant :

```
untie attempted while 1 inner references still exist
```

Pour faire fonctionner correctement ce script et sans message, assurez-vous qu'il n'existe plus de références valides vers l'objet lié *avant* d'appeler untie() :

```
undef $x;
untie $fred;
```

Maintenant que UNTIE existe, le concepteur de la classe peut décider quelle partie des fonctionnalités doit être associée à un untie et celle qui doit être associée à la destruction de l'objet. Le choix de ce qui doit être conservé pour une classe donnée dépend des appels à aux méthodes ne relevant pas de la liaison et qui doivent pouvoir fonctionner sur l'objet lui-même. Dans la plupart des cas, il est maintenant recommandé de placer dans la méthode UNTIE tout ce qui était précédemment dans DESTROY.

Si la méthode UNTIE existe, l'avertissement ci-dessus ne peut pas avoir lieu. À la place, la méthode UNTIE reçoit comme argument le compte des références existantes et peut émettre son propre avertissement si nécessaire. Pour simuler le comportement correspondant au cas sans méthode UNTIE, la méthode suivante peut être utilisée :

```

sub UNTIE
{
 my ($obj,$count) = @_;
 carp "untie attempted while $count inner references still exist" if $count;
}

```

## 43.3 VOIR AUSSI

Voir `DB_File` ou `Config` pour quelques exemples d'utilisations intéressantes de `tie()`. Une bonne base de départ pour la plupart des utilisations de `tie()` passe par l'un des modules `Tie::Scalar`, `Tie::Array`, `Tie::Hash` ou `Tie::Handle`.

## 43.4 BUGS

L'information d'utilisation de "buckets" fournie par `scalar(%hash)` n'est pas disponible. Cela signifie que son utilisation dans un contexte booléen est impossible (actuellement, cela retourne toujours faux que la table de hachage soit vide ou contienne des éléments).

La localisation de tableaux ou de tables de hachage liés ne fonctionne pas. Au moment où on quitte la portée de l'appel à local, le tableau ou la table ne retrouve pas sa valeur initiale.

Compter le nombre d'entrées d'une table de hachage via `scalar(keys(%hash))` ou `scalar(values(%hash))` est très inefficace puisque cela nécessite un parcours complet de la table via `FIRSTKEY/NEXTKEY`.

La gestion des tranches (slices) d'une table ou d'un tableau lié fait appel à de nombreux appels `FETCH/STORE` puisqu'il n'y a pas de méthodes liées pour les opérations sur les tranches.

Vous ne pouvez pas lier facilement une structure multi-niveaux (comme une table de tables de hachage) à un fichier `dbm`. Le premier problème est la limite de taille d'une valeur (sauf pour `GDBM` et `Berkeley DB`). Mais, au-delà, vous aurez aussi le problème du stockage de références sur disque. L'un des modules expérimentaux qui tente de répondre à ce besoin est `DBM::Deep`. Allez voir le code source sur un site CPAN (comme décrit par *perlmodlib*). Notez que, contrairement à ce que pourrait laisser croire son nom, `DBM::Deep` n'utilise pas `dbm`. Un autre module à voir est `MLDBM`, qui est aussi sur CPAN, mais il a de nombreuses limitations.

Les descripteurs de fichier liés sont encore incomplets. Les appels à `sysopen()`, `truncate()`, `flock()`, `fcntl()`, `stat()` et `-X` ne peuvent pas encore être interceptés.

## 43.5 AUTEURS

Tom Christiansen

`TIEHANDLE` par Sven Verdoolaege <[skimo@dns.ufsia.ac.be](mailto:skimo@dns.ufsia.ac.be)> et Doug MacEachern <[dougm@osf.org](mailto:dougm@osf.org)>

`UNTIE` par Nick Ing-Simmons <[nick@ing-simmons.net](mailto:nick@ing-simmons.net)>

`SCALAR` par Tassilo von Parseval <[tassilo.von.parseval@rwth-aachen.de](mailto:tassilo.von.parseval@rwth-aachen.de)>

Les tableaux liés par Casey West <[casey@geeknest.com](mailto:casey@geeknest.com)>

## 43.6 TRADUCTION

### 43.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec `perl 5.10.0`. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 43.6.2 Traducteur

Traduction initiale et mise à jour `v5.10.0` : Paul Gaborit <[Paul.Gaborit@enstimac.fr](mailto:Paul.Gaborit@enstimac.fr)>

### 43.6.3 Relecture

Régis Julié <[Regis.Julie@cetelem.fr](mailto:Regis.Julie@cetelem.fr)>



# Chapitre 44

## perlipc

Communication inter-processus en Perl (signaux, files d'attente, tubes, sous-processus sûrs, sockets et sémaphores)

### 44.1 DESCRIPTION

Les équipements CIP (IPC en anglais, NDT) de base de Perl sont construits sur les signaux, tubes nommés, tubes ouverts, routines de socket de Berkeley et appels CIP du System V, du bon vieux Unix. Chacun est utilisé dans des situations légèrement différentes.

### 44.2 Signaux

Perl utilise un modèle simple de manipulation des signaux : le hachage %SIG contient les noms des handlers (gestionnaires) de signaux installés par l'utilisateur, ou des références vers eux. Ces handlers seront appelés avec un argument qui est le nom du signal qui l'a déclenché. Un signal peut être généré intentionnellement par une séquence particulière au clavier comme control-C ou control-Z, ou vous être envoyé par un autre processus, ou être déclenché automatiquement par le noyau lorsque des événements spéciaux se produisent, comme la fin d'un processus fils, ou l'épuisement de l'espace disponible sur la pile de votre processus, ou la rencontre d'une limite sur la taille d'un fichier.

Par exemple, pour capturer un signal d'interruption, installez un handler comme ceci. Faites aussi peu de choses que possible dans votre handler ; notez comment nous nous débrouillons pour ne faire que placer une variable globale et lever une exception. C'est parce que sur la plupart des systèmes, les bibliothèques ne sont pas réentrantes, en particulier l'allocation de mémoire et les routines d'entrée/sortie. Cela signifie que pratiquement *quoi que ce soit* dans votre handler pourrait en théorie provoquer une faute de segmentation et par conséquent un coredump.

```
sub catch_zap {
 my $signame = shift;
 $shucks++;
 die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap'; # pourrait echouer dans un module
$SIG{INT} = \&catch_zap; # meilleur strategie
```

Les noms des signaux sont ceux qui sont listés par `kill -l` sur votre système, ou que vous pouvez obtenir via le module Config. Installez une liste @signame indexée par numéro pour avoir le nom et une table %signo indexée par le nom pour avoir le numéro :

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
 $signo{$name} = $i;
 $signame[$i] = $name;
 $i++;
}
```

Donc pour vérifier si le signal 17 et SIGALRM sont les mêmes, faites juste ceci :

```
print "signal #17 = $signalname[17]\n";
if ($signo{ALRM}) {
 print "SIGALRM is $signo{ALRM}\n";
}
```

Vous pouvez aussi choisir d'affecter les chaînes 'IGNORE' ou 'DEFAULT' comme handler, dans ce cas Perl essaiera d'ignorer le signal ou de réaliser l'action par défaut.

Sur la plupart des plateformes Unix, le signal CHLD (parfois aussi connu sous le nom CLD) a un comportement spécial en fonction de la valeur 'IGNORE'. Mettre \$SIG{CHLD} à 'IGNORE' sur une telle plateforme a pour effet de ne pas créer de processus zombie lorsque le processus père échoue dans l'attente (wait()) de son fils (i.e. les processus fils sont automatiquement détruits). Appeler wait() avec \$SIG{CHLD} placé à 'IGNORE' retourne habituellement -1 sur de telles plateformes.

Certains signaux ne peuvent être ni piégés ni ignorés, comme les signaux KILL et STOP (mais pas TSTP). Une stratégie pour ignorer temporairement des signaux est d'utiliser une déclaration local(), qui sera automatiquement restaurée à la sortie de votre bloc d'instructions (Souvenez-vous que les valeurs local() sont "héritées" par les fonctions appelées depuis ce bloc).

```
sub precious {
 local $SIG{INT} = 'IGNORE';
 &more_functions;
}
sub more_functions {
 # interrupts still ignored, for now...
}
```

Envoyer un signal à un identifiant de processus négatif signifie que vous envoyez ce signal à tout le groupe de processus Unix. Ce code envoie un signal hang-up à tous les processus du groupe courant (et place \$SIG{HUP} sur IGNORE pour qu'il ne se tue pas lui-même) :

```
{
 local $SIG{HUP} = 'IGNORE';
 kill HUP => -$$;
 # snazzy writing of: kill('HUP', -$$)
}
```

Un autre signal intéressant à envoyer est le signal numéro zéro. Il n'affecte en vérité aucun autre processus, mais vérifie s'il est encore en vie ou a changé son UID.

```
unless (kill 0 => $kid_pid) {
 warn "something wicked happened to $kid_pid";
}
```

Vous pourriez aussi vouloir employer des fonctions anonymes comme handlers de signaux simples :

```
$SIG{INT} = sub { die "\nOutta here!\n" };
```

Mais cela sera problématique pour les handlers plus compliqués qui ont besoin de se réinstaller eux-mêmes. Puisque le mécanisme de signalisation de Perl est basé sur la fonction signal(3) de la bibliothèque C, vous pourriez parfois avoir la malchance d'utiliser des systèmes où cette fonction est "cassée", c'est-à-dire qu'elle se comporte selon l'ancienne façon peu fiable du SysV plutôt que la plus récente et plus raisonnable méthode BSD et POSIX. Vous verrez donc les gens sur la défensive écrire des handlers de signaux ainsi :

```
sub REAPER {
 $waitedpid = wait;
 # maudissez sysV : il nous force non seulement a
 # réinstaller le handler, mais aussi a le placer
 # apres le wait
 $SIG{CHLD} = \&REAPER;
}
$SIG{CHLD} = \&REAPER;
maintenant, on fait quelque chose qui forke...
```

Ou même de cette façon plus élaborée :

```
use POSIX ":sys_wait_h";
sub REAPER {
 my $child;
 while (($child = waitpid(-1,WNOHANG)) {
 $Kid_Status{$child} = $?;
 }
 $SIG{CHLD} = \&REAPER; # maudissez encore sysV
}
$SIG{CHLD} = \&REAPER;
on fait quelque chose qui forke...
```

La manipulation de signaux est aussi utilisée pour les timeouts sous Unix. Pendant que vous êtes soigneusement protégé par un bloc `eval{}`, vous installez un handler de signal pour piéger les signaux d'alarme puis programmez de vous en faire délivrer un dans un certain nombre de secondes. Puis vous essayez votre opération bloquante, annulant l'alarme lorsque c'est fait mais pas avant d'être sorti de votre bloc `eval{}`. Si cela ne marche pas, vous utiliserez `die()` pour sauter hors du bloc, tout comme vous utiliseriez `longjmp()` ou `throw()` dans d'autres langages.

Voici un exemple :

```
eval {
 local $SIG{ALRM} = sub { die "alarm clock restart" };
 alarm 10;
 flock(FH, 2); # lock d'écriture blocante
 alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { die }
```

Si l'opération chronométrée est `system()` ou `qX()`, cette technique peut générer des zombies. Si cela vous préoccupe, vous devrez faire vos propres `fork()` et `exec()`, et tuer les processus fils errants.

Pour une manipulation plus complexe des signaux, vous pourriez voir le module du standard POSIX. Il est lamentable que Ceci soit presque entièrement non documenté, mais le fichier `/lib/posix.t` de la distribution source de Perl contient quelques exemples.

## 44.3 Tubes nommés

Un tube nommé (souvent appelé une file d'attente FIFO) est un vieux mécanisme de CIP Unix pour les processus communicant sur la même machine. Il fonctionne tout comme un tube anonyme, normal et connecté, sauf que les processus se donnent rendez-vous en utilisant un nom de fichier et n'ont pas besoin d'être apparentés.

Pour créer un tube nommé, utilisez la commande Unix `mknod(1)` ou sur certains systèmes, `mkfifo(1)`. Elles peuvent ne pas se trouver dans votre chemin d'accès normal.

```
le systeme renvoie les vals a l'envers, donc && et pas ||
#
$ENV{PATH} .= ":/etc:/usr/etc";
if (system('mknod', $path, 'p')
 && system('mkfifo', $path))
{
 die "mk{nod,fifo} $path failed";
}
```

Une FIFO est pratique quand vous voulez connecter un processus à un autre sans qu'ils soient apparentés. Lorsque vous ouvrez une FIFO, le programme se bloque jusqu'à ce que quelque chose arrive à l'autre bout.

Par exemple, disons que vous aimeriez que votre fichier `.signature` soit un tube nommé connecté à un programme en Perl. Désormais, chaque fois qu'un programme (comme un gestionnaire de courrier électronique, un lecteur de news, un finger, etc.) essaye de lire le contenu de ce fichier, il se bloque et votre programme lui fournit une nouvelle signature. Nous utiliserons le test de fichier `-p` qui teste un tube pour déterminer si quelqu'un (ou quelque chose) a accidentellement supprimé notre FIFO.

```

chdir; # a la maison
$FIFO = '.signature';
$ENV{PATH} .= ":/etc:/usr/games";

while (1) {
 unless (-p $FIFO) {
 unlink $FIFO;
 system('mknod', $FIFO, 'p')
 && die "can't mknod $FIFO: $!";
 }

 # la ligne suivante bloque jusqu'a ce qu'il y ait un lecteur
 open (FIFO, "> $FIFO") || die "can't write $FIFO: $!";
 print FIFO "John Smith (smith\@host.org)\n", 'fortune -s';
 close FIFO;
 sleep 2; # pour eviter les signaux dupliques
}

```

### 44.3.1 AVERTISSEMENT

En installant du code Perl qui traite des signaux, vous vous exposez à deux types de dangers. D'abord, peu de fonctions de la bibliothèque système sont réentrantes. Si le signal provoque une interruption pendant que Perl exécute une fonction (comme `malloc(3)` ou `printf(3)`), et que votre handler de signal appelle alors la même fonction, vous pourriez obtenir un comportement non prévisible - souvent, un core dump. Ensuite, Perl n'est pas lui-même réentrant aux niveaux les plus bas. Si le signal interrompt Perl pendant qu'il change ses propres structures de données internes, un comportement non prévisible similaire peut apparaître.

Vous pouvez faire deux choses, suivant que vous êtes paranoïaque ou pragmatique. L'approche paranoïaque est d'en faire aussi peu que possible dans votre handler. Fixez une variable entière existant et ayant déjà une valeur et revenez. Ceci ne vous aide pas si vous êtes au milieu d'un appel système lent, qui se contentera de recommencer. Cela veut dire que vous devez mourir avec `die` pour sauter (`longjump(3)`) hors du handler. Même ceci est quelque peu cavalier pour le véritable paranoïaque, qui évite `die` dans un handler car le système *est* là pour vous attraper. L'approche pragmatique consiste à dire "je connais les risques, mais je préfère la facilité", et ensuite faire tout ce qu'on veut dans le handler de signal, en étant prêt à nettoyer les core-dumps de temps en temps.

Interdire totalement les handlers interdirait aussi beaucoup de programmes intéressants, y compris virtuellement tout ce qui se trouve dans cette page de manuel, puisque vous ne pourriez plus écrire ne serait-ce qu'un handler `SIGCHLD`. Ce problème épineux devrait être réglé dans la version 5.005.

## 44.4 Utilisation de open() pour la CIP

L'instruction `open()` de base en Perl peut aussi être utilisée pour la communication inter-processus unidirectionnelle en ajoutant un symbole de tube juste avant ou après le second argument de `open()`. Voici comment démarrer un processus fils dans lequel vous avez l'intention d'écrire :

```

open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
 || die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER || die "bad spool: $! $?";

```

Et voici comment démarrer un processus fils depuis lequel vous désirez lire :

```

open(STATUS, "netstat -an 2>&1 |")
 || die "can't fork: $!";
while (<STATUS>) {
 next if /^(tcp|udp)/;
 print;
}
close STATUS || die "bad netstat: $! $?";

```

Si l'on peut être certain qu'un programme spécifique est un script Perl qui attend des noms de fichiers dans @ARGV, le programmeur futé peut écrire quelque chose comme ceci :

```
% program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

et, indépendamment du shell d'où il est appelé, le programme Perl lira dans le fichier *f1*, le processus *cmd1*, l'entrée standard (*tmpfile* dans ce cas), le fichier *f2*, la commande *cmd2*, et finalement dans le fichier *f3*. Plutôt débrouillard, hein ?

Vous pourriez remarquer que l'on pourrait utiliser des accents graves pour obtenir à peu près le même effet que l'ouverture d'un tube en lecture :

```
print grep { !/^(tcp|udp)/ } 'netstat -an 2>&1';
die "bad netstat" if $?;
```

Même si ceci est vrai en surface, il est bien plus efficace de traiter le fichier une ligne ou un enregistrement à la fois car alors vous n'avez pas à charger toute la chose en mémoire d'un seul coup. Cela vous donne aussi un contrôle plus fin sur tout le processus, vous laissant tuer le processus fils plus tôt si vous le voulez.

Prenez soin de vérifier à la fois les valeurs de retour de `open()` et de `close()`. Si vous *écrivez* dans un tube, vous pouvez aussi piéger SIGPIPE. Sinon, pensez à ce qui se passe lorsque vous ouvrez un tube vers une commande qui n'existe pas : le `open()` réussira très probablement (il ne fait que refléter le succès du `fork()`), mais ensuite votre sortie échouera - spectaculairement. Perl ne peut pas savoir si la commande a fonctionné parce que votre commande tourne en fait dans un processus séparé dont l'`exec()` peut avoir échoué. Par conséquent, ceux qui lisent depuis une commande bidon retournent juste une rapide fin de fichier, ceux qui écrivent vers une commande bidon provoqueront un signal qu'ils feraient mieux d'être préparés à gérer. Considérons :

```
open(FH, "|bogus") or die "can't fork: $!";
print FH "bang\n" or die "can't write: $!";
close FH or die "can't close: $!";
```

Ceci n'explosera pas avant le `close`, et ce sera sur un SIGPIPE. Pour l'intercepter, vous pourriez utiliser ceci :

```
$_SIG{PIPE} = 'IGNORE';
open(FH, "|bogus") or die "can't fork: $!";
print FH "bang\n" or die "can't write: $!";
close FH or die "can't close: status=$?";
```

#### 44.4.1 Handles de Fichiers

Le processus principal et tous les processus fils qu'il peut générer partagent les mêmes handles de fichiers STDIN, STDOUT et STDERR. Si deux processus essaient d'y accéder en même temps, des choses étranges peuvent se produire. Vous pourriez aussi fermer ou réouvrir les handles de fichiers pour le fils. Vous pouvez contourner cela en ouvrant votre tube avec `open()`, mais sur certains systèmes cela signifie que le processus fils ne peut pas survivre à son père.

#### 44.4.2 Processus en Arrière-Plan.

Vous pouvez exécuter une commande en arrière-plan avec :

```
system("cmd &");
```

Les STDOUT et STDERR de la commande (et peut-être STDIN, selon votre shell) seront les mêmes que ceux du père. Vous n'aurez pas besoin de piéger SIGCHLD à cause du double `fork` qui se produit (voir plus bas pour plus de détails).

### 44.4.3 Dissociation Complète du Fils et de son Père

Dans certains cas (démarrage de processus serveurs, par exemple) vous voudrez complètement dissocier le processus fils de son père. Ceci est souvent appelé une démonisation. Un démon bien élevé fera aussi un `chdir()` vers le répertoire `root` (pour qu'il n'empêche pas le démontage du système de fichiers contenant le répertoire à partir duquel il a été lancé) et redirige ses descripteurs de fichier standard vers `/dev/null` (pour que des sorties aléatoires ne finissent pas sur le terminal de l'utilisateur).

```
use POSIX 'setsid';

sub daemonize {
 chdir '/' or die "Can't chdir to /: $!";
 open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
 open STDOUT, '>/dev/null'
 or die "Can't write to /dev/null: $!";
 defined(my $pid = fork) or die "Can't fork: $!";
 exit if $pid;
 setsid or die "Can't start a new session: $!";
 open STDERR, '>&STDOUT' or die "Can't dup stdout: $!";
}

```

Le `fork()` doit venir avant le `setsid()` pour s'assurer que vous n'êtes pas un leader de groupe de processus (le `setsid()` échouera si vous l'êtes). Si votre système ne possède pas la fonction `setsid()`, ouvrez `/dev/tty` et utilisez dessus l'`ioctl()` `TIOCNOTTY` à la place. Voir `tty(4)` pour plus de détails.

Les utilisateurs non unixiens devraient jeter un oeil sur leur module `Votre_OS::Process` pour avoir d'autres solutions.

### 44.4.4 Ouvertures Sûres d'un Tube

Une autre approche intéressante de la CIP est de rendre votre simple programme multiprocessus et de communiquer entre (ou même parmi) eux. La fonction `open()` acceptera un fichier en argument pour `"|"` ou `"|-"` afin de faire une chose très intéressante : elle forkera un fils connecté au descripteur de fichier que vous venez d'ouvrir. Le fils exécute le même programme que le parent. C'est utile par exemple pour ouvrir de façon sécurisée un fichier lorsque l'on s'exécute sous un UID ou un GID présumé. Si vous ouvrez un tube *vers* minus, vous pouvez écrire dans le descripteur de fichier que vous avez ouvert et votre fils le trouvera dans son `STDIN`. Si vous ouvrez un tube *depuis* minus, vous pouvez lire depuis le descripteur de fichier tout ce que votre fils écrit dans son `STDOUT`.

```
use English;
my $sleep_count = 0;

do {
 $pid = open(KID_TO_WRITE, "|-");
 unless (defined $pid) {
 warn "cannot fork: $!";
 die "bailing out" if $sleep_count++ > 6;
 sleep 10;
 }
} until defined $pid;

if ($pid) { # parent
 print KID_TO_WRITE @some_data;
 close(KID_TO_WRITE) || warn "kid exited $?";
} else { # child
 ($EUID, $EGID) = ($UID, $GID); # suid progs only
 open (FILE, "> /safe/file")
 || die "can't open /safe/file: $!";
 while (<STDIN) {
 print FILE; # child's STDIN is parent's KID
 }
 exit; # n'oubliez pas ceci
}

```

Un autre usage courant de cette construction apparaît lorsque vous avez besoin d'exécuter quelque chose sans interférences de la part du shell. Avec `system()`, c'est direct, mais vous ne pouvez pas utiliser un tube ouvert ou des accents graves de façon sûre. C'est parce qu'il n'y a pas de moyen d'empêcher le shell de mettre son nez dans vos arguments. À la place, utilisez le contrôle de bas niveau pour appeler `exec()` directement.

Voici un backtick ou un tube ouvert en lecture sûrs :

```
ajoutez un traitement d'erreur comme ci-dessus
$pid = open(KID_TO_READ, "-|");

if ($pid) { # parent
 while (<KID_TO_READ>) {
 # faites quelque chose d'intéressant
 }
 close(KID_TO_READ) || warn "kid exited $?";
} else { # fils
 ($EUID, $EGID) = ($UID, $GID); # suid uniquement
 exec($program, @options, @args)
 || die "can't exec program: $!";
 # PASATTEINT
}
```

Et voici un tube sûr ouvert en écriture :

```
ajoutez un traitement d'erreur comme ci-dessus
$pid = open(KID_TO_WRITE, "|-");
$SIG{ALRM} = sub { die "whoops, $program pipe broke" };

if ($pid) { # parent
 for (@data) {
 print KID_TO_WRITE;
 }
 close(KID_TO_WRITE) || warn "kid exited $?";
} else { # fils
 ($EUID, $EGID) = ($UID, $GID);
 exec($program, @options, @args)
 || die "can't exec program: $!";
 # PASATTEINT
}
```

Notez que ces opérations sont des forks Unix complets, ce qui signifie qu'ils peuvent ne pas être correctement implémentés sur des systèmes étrangers. De plus, il ne s'agit pas d'un vrai multithreading. Si vous désirez en apprendre plus sur le threading, voyez le fichier *modules* mentionnée plus bas dans la section VOIR AUSSI.

#### 44.4.5 Communication Bidirectionnelle avec un autre Processus

Même si ceci fonctionne raisonnablement bien pour la communication unidirectionnelle, qu'en est-il pour la communication bidirectionnelle ? La chose la plus évidente que vous aimeriez faire ne marche en fait pas :

```
open(PROG_FOR_READING_AND_WRITING, "| un programme |")
```

et si vous oubliez d'utiliser le pragma `use warnings` ou l'option `-w`, alors vous raterez complètement le message de diagnostic :

```
Can't do bidirectional pipe at -e line 1.
```

Si vous le voulez vraiment, vous pouvez utiliser la fonction de la bibliothèque standard `open2()` pour attraper les deux bouts. Il existe aussi une `open3()` pour les E/S tridirectionnelles de façon que vous puissiez aussi récupérer le `STDERR` de votre fils, mais faire ceci nécessiterait une boucle `select()` maladroite et ne vous permettrait pas d'utiliser les opérations d'entrée normales de Perl.

Si vous jetez un oeil sur son source, vous verrez que `open2()` utilise des primitives de bas niveau, comme les appels `pipe()` et `exec()` d'Unix, pour créer toutes les connexions. Il aurait peut-être été un peu plus efficace d'utiliser `socketpair()`, mais cela serait devenu encore moins portable que ce ne l'est déjà. Les fonctions `open2()` et `open3()` ont peu de chances de fonctionner ailleurs que sur un système Unix ou quelques autres prétendant être conformes à la norme POSIX.

Voici un exemple d'utilisation de `open2()`:

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -u -n");
print Writer "stuff\n";
$got = <Reader>;
```

Le problème avec ceci est que le buffering d'Unix va vraiment rendre cette opération pénible. Même si votre descripteur de fichier `Writer` est vidé automatiquement, et même si le processus à l'autre bout reçoit vos données de façon régulière, vous ne pouvez habituellement rien faire pour le forcer à vous les renvoyer rapidement de la même façon. Dans ce cas-ci, nous le pouvons, car nous avons donné à `cat` une option `-u` pour qu'il n'ait pas de tampon. Mais très peu de commandes Unix sont conçues pour fonctionner à travers des tubes, ceci marche donc rarement à moins que vous ayez écrit vous-mêmes le programme se trouvant à l'autre bout du tube à deux sens.

Une solution à ceci est la bibliothèque non standard *Comm.pl*. Elle utilise des pseudo-ttys pour rendre le comportement de votre programme plus raisonnable :

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
 print $ph "a line\n";
 print "got back ", scalar <$ph>;
}
```

De cette façon vous n'avez pas besoin de contrôler le code source du programme que vous utilisez. La bibliothèque *Comm* contient aussi les fonctions `expect()` et `interact()`. Vous trouverez la bibliothèque (et, nous l'espérons, son successeur *IPC::Chat*) dans l'archive CPAN la plus proche de vous, comme il est détaillé dans la section VOIR AUSSI ci-dessous.

Le module plus récent *Expect.pm* sur le CPAN s'occupe aussi de ce genre de choses. Ce module nécessite deux autres modules du CPAN : *IO::Pty* et *IO::Stty*. Il installe un pseudo-terminal pour interagir avec les programmes qui insistent pour discuter avec le pilote de périphérique du terminal. Si votre système fait partie de ceux supportés, cela pourrait être la meilleure solution pour vous.

#### 44.4.6 Communication Bidirectionnelle avec Vous-même

Si vous voulez, vous pouvez faire des `pipe()` et des `fork()` de bas niveau pour coudre tout cela ensemble à la main. Cet exemple ne se parle qu'à lui-même, mais vous pourriez réouvrir les handles appropriés vers `STDIN` et `STDOUT` et appeler d'autres processus.

```
#!/usr/bin/perl -w
pipe1 - communication bidirectionnelle utilisant deux paires
de tubes concus pour les systèmes n'ayant pas l'appel socketpair()
use IO::Handle; # milliers de lignes juste pour l'autoflush :-(
pipe(PARENT_RDR, CHILD_WTR); # XXX: echec ?
pipe(CHILD_RDR, PARENT_WTR); # XXX: echec ?
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);

if ($pid = fork) {
 close PARENT_RDR; close PARENT_WTR;
 print CHILD_WTR "Parent Pid $$ is sending this\n";
 chomp($line = <CHILD_RDR>);
```



```

 print "Parent Pid $$ just read this: '$line'\n";
 close CHILD_RDR; close CHILD_WTR;
 waitpid($pid,0);
} else {
 die "cannot fork: $!" unless defined $pid;
 close CHILD_RDR; close CHILD_WTR;
 chomp($line = <PARENT_RDR>);
 print "Child Pid $$ just read this: '$line'\n";
 print PARENT_WTR "Child Pid $$ is sending this\n";
 close PARENT_RDR; close PARENT_WTR;
 exit;
}

```

Mais vous n'avez en fait pas besoin de faire deux appels à des tubes. Si vous disposez de l'appel système `socketpair()`, il fera tout cela pour vous.

```

#!/usr/bin/perl -w
pipe2 - communication bidirectionnelle utilisant socketpair
"les choses partagées dans les deux sens sont toujours les meilleures"

use Socket;
use IO::Handle; # milliers de lignes juste pour l'autoflush :-()
Nous utilisons AF_UNIX car bien que *_LOCAL est la forme
POSIX 1003.1g de la constante, beaucoup de machines ne
l'ont pas encore.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
 or die "socketpair: $!";

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork) {
 close PARENT;
 print CHILD "Parent Pid $$ is sending this\n";
 chomp($line = <CHILD>);
 print "Parent Pid $$ just read this: '$line'\n";
 close CHILD;
 waitpid($pid,0);
} else {
 die "cannot fork: $!" unless defined $pid;
 close CHILD;
 chomp($line = <PARENT>);
 print "Child Pid $$ just read this: '$line'\n";
 print PARENT "Child Pid $$ is sending this\n";
 close PARENT;
 exit;
}

```

## 44.5 Sockets : Communication Client/Serveur

Même si elles ne sont pas limitées aux systèmes d'exploitation dérivés d'Unix (e.g., WinSock sur les PC fournit le support des sockets, tout comme le font les bibliothèques VMS), vous ne disposez peut-être pas des sockets sur votre système, auquel cas cette section ne vous fera probablement pas beaucoup de bien. Avec les sockets, vous pouvez à la fois créer des circuits virtuels (i.e., des streams TCP) et des datagrammes (i.e., des paquets UDP). Vous pouvez même peut-être faire plus, en fonction de votre système.

Les appels Perl pour traiter les sockets ont les mêmes noms que les appels système correspondants en C, mais leurs arguments tendent à être différents pour deux raisons : tout d'abord, les handles de fichiers de Perl fonctionnent différemment des descripteurs de fichiers en C. Ensuite, Perl connaît déjà la longueur de ses chaînes, vous n'avez donc pas besoin de passer cette information.

L'un des problèmes majeurs avec l'ancien code des sockets de Perl était qu'il utilisait des valeurs codées en dur pour certaines des constantes, ce qui endommageait sévèrement la portabilité. Si vous avez déjà vu du code qui fait des choses comme fixer explicitement `$AF_INET = 2`, vous savez que vous avez un gros problème : une approche incommensurablement supérieure est d'utiliser le module `Socket`, qui fournit un accès plus fiable aux différentes constantes et fonctions dont vous aurez besoin.

Si vous n'êtes pas en train d'écrire un couple serveur/client pour un protocole existant comme NNTP or SMTP, vous devriez réfléchir à la façon dont votre serveur saura quand le client a fini de parler, et vice-versa. La plupart des protocoles sont basés sur des messages et des réponses d'une ligne (de façon que l'une des parties sache que l'autre a fini quand un `"\n"` est reçu) ou sur des messages et des réponses de plusieurs lignes qui se finissent par un point ou une ligne vide (`"\n.\n"` termine un message ou une réponse).

### 44.5.1 Terminateur de Ligne Internet

Le terminateur de ligne de l'Internet est `"\015\012"`. Sous certaines variantes ASCII d'Unix, cela peut habituellement être écrit `"\r\n"`, mais sous certains autres systèmes, `"\r\n"` peut certaines fois être `"\015\015\012"`, `"\012\012\015"`, ou quelque chose de complètement différent. Les standards spécifient d'écrire `"\015\012"` pour être conforme (soyez strict sur ce que vous fournissez), mais ils recommandent aussi d'accepter un `"\012"` solitaire en entrée (soyez indulgent sur ce que vous attendez). Nous n'avons pas toujours été très bon à ce sujet dans le code de cette page de manuel, mais à moins que vous ne soyez sur un Mac, cela devrait aller.

### 44.5.2 Clients et Serveurs TCP Internet

Utilisez des sockets Internet lorsque vous voulez effectuer une communication client-serveur qui pourrait s'étendre à des machines hors de votre propre système.

Voici un exemple de client TCP utilisant des sockets Internet :

```
#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote,$port, $iaddr, $paddr, $proto, $line);

$remote = shift || 'localhost';
$port = shift || 2345; # port pris au hasard
if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }
die "No port" unless $port;
$iaddr = inet_aton($remote) || die "no host: $remote";
$paddr = sockaddr_in($port, $iaddr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
connect(SOCK, $paddr) || die "connect: $!";
while (defined($line = <SOCK>)) {
 print $line;
}

close (SOCK) || die "close: $!";
exit;
```

Et voici le serveur correspondant pour l'accompagner. Nous laisserons l'adresse sous la forme `INADDR_ANY` pour que le noyau puisse choisir l'interface appropriée sur les hôtes à plusieurs pattes. Si vous voulez rester sur une interface particulière (comme le côté externe d'une passerelle ou d'un firewall), vous devriez la remplacer par votre véritable adresse.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
$EOL = "\015\012";
```

```

sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
$port = $1 if $port =~ /(\d+)/; # nettoie le numero de port

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
 pack("l", 1)) || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $paddr;

$SIG{CHLD} = \&REAPER;

for (; $paddr = accept(Client,Server); close Client) {
 my($port,$iaddr) = sockaddr_in($paddr);
 my $name = gethostbyaddr($iaddr,AF_INET);

 logmsg "connection from $name [",
 inet_ntoa($iaddr), "]"
 at port $port";

 print Client "Hello there, $name, it's now ",
 scalar localtime, $EOL;
}

```

Et voici une version multithread. Elle est multithread en ce sens que comme la plupart des serveurs, elle génère (fork) un serveur esclave pour manipuler la requête du client de façon que le serveur maître puisse rapidement revenir servir un nouveau client.

```

#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
$EOL = "\015\012";

sub spawn; # déclaration préalable
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
$port = $1 if $port =~ /(\d+)/; # nettoie le numéro de port

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
 pack("l", 1)) || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

```

```

sub REAPER {
 $waitpid = wait;
 $SIG{CHLD} = \&REAPER; # abhorrons sysV
 logmsg "reaped $waitpid" . ($? ? " with exit $" : '');
}

$SIG{CHLD} = \&REAPER;

for ($waitpid = 0;
 ($paddr = accept(Client,Server)) || $waitpid;
 $waitpid = 0, close Client)
{
 next if $waitpid and not $paddr;
 my($port,$iaddr) = sockaddr_in($paddr);
 my $name = gethostbyaddr($iaddr,AF_INET);

 logmsg "connection from $name [",
 inet_ntoa($iaddr), "]"
 at port $port";

 spawn sub {
 print "Hello there, $name, it's now ", scalar localtime, $EOL;
 exec '/usr/games/fortune' # XXX: 'mauvais' terminateurs de ligne
 or confess "can't exec fortune: $!";
 };
}

sub spawn {
 my $coderef = shift;

 unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
 confess "usage: spawn CODEREF";
 }

 my $pid;
 if (!defined($pid = fork)) {
 logmsg "cannot fork: $!";
 return;
 } elsif ($pid) {
 logmsg "begat $pid";
 return; # Je suis le pere
 }
 # ou bien je suis le fils - on se multiplie

 open(STDIN, "<&Client") || die "can't dup client to stdin";
 open(STDOUT, ">&Client") || die "can't dup client to stdout";
 ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
 exit &$coderef();
}

```

Ce serveur prend la peine de cloner un processus fils via `fork()` pour chaque requête entrante. De cette façon, il peut répondre à de nombreuses requêtes en même temps, ce que vous pourriez ne pas toujours désirer. Même si vous n'utilisez pas `fork()`, le `listen()` permettra de nombreuses connexions en cours. Les serveurs qui se dupliquent doivent prendre tout particulièrement soin de nettoyer leurs enfants morts (appelés "zombies" en jargon unixien), car autrement vous remplirez rapidement votre table des processus.

Nous vous suggérons d'utiliser l'option `-T` pour profiter du taint checking (voir *perlsec*) même si vous ne tournez pas en `setuid` ou en `setgid`. C'est toujours une bonne idée pour les serveurs et les programmes exécutés au nom de quelqu'un d'autre (comme les scripts CGI), car cela diminue le risque que des personnes extérieures compromettent votre système.

Étudions un autre client TCP. Celui-ci se connecte au service TCP "time" sur de nombreuses machines différentes et montre à quel point leurs horloges diffèrent du système sur lequel il est exécuté :

```
#!/usr/bin/perl -w
use strict;
use Socket;

my $SECS_of_70_YEARS = 2208988800;
sub ctime { scalar localtime(shift) }

my $iaddr = gethostbyname('localhost');
my $proto = getprotobyname('tcp');
my $port = getservbyname('time', 'tcp');
my $paddr = sockaddr_in(0, $iaddr);
my($host);

$| = 1;
printf "%-24s %8s %s\n", "localhost", 0, ctime(time());

foreach $host (@ARGV) {
 printf "%-24s ", $host;
 my $hisiaddr = inet_pton($host) || die "unknown host";
 my $hispaddr = sockaddr_in($port, $hisiaddr);
 socket(SOCKET, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
 connect(SOCKET, $hispaddr) || die "bind: $!";
 my $rtime = ' ';
 read(SOCKET, $rtime, 4);
 close(SOCKET);
 my $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
 printf "%8d %s\n", $histime - time, ctime($histime);
}

```

### 44.5.3 Clients et Serveurs TCP du Domaine Unix

C'est bien pour les clients et les serveurs Internet, mais pour les communications locales ? Même si vous pouvez utiliser la même configuration, vous ne le voulez pas toujours. Les sockets du domaine Unix sont locales à l'hôte courant, et sont souvent utilisées en interne pour implémenter des tubes. Contrairement aux sockets du domaine Internet, les sockets Unix peuvent se voir dans le système de fichier par `ls(1)`.

```
% ls -l /dev/log
srw-rw-rw- 1 root 0 Oct 31 07:23 /dev/log
```

Vous pouvez les tester avec le test de fichier `-S` de Perl :

```
unless (-S '/dev/log') {
 die "something's wicked with the print system";
}

```

Voici un exemple de client du domaine Unix :

```
#!/usr/bin/perl -w
use Socket;
use strict;
my ($rendezvous, $line);

$rendezvous = shift || '/tmp/catsock';
socket(SOCK, PF_UNIX, SOCK_STREAM, 0) || die "socket: $!";
connect(SOCK, sockaddr_un($rendezvous)) || die "connect: $!";
while (defined($line = <SOCK>)) {
 print $line;
}
exit;

```

Et voici un serveur correspondant. Vous n'avez pas besoin de vous soucier ici des stupides terminateurs de réseau car les sockets Unix sont de façon certaine sur l'hôte local (localhost, NDT), et ainsi tout fonctionne bien.

```
#!/usr/bin/perl -Tw
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $NAME = '/tmp/catsock';
my $uaddr = sockaddr_un($NAME);
my $proto = getprotobyname('tcp');

socket(Server,PF_UNIX,SOCK_STREAM,0) || die "socket: $!";
unlink($NAME);
bind (Server, $uaddr) || die "bind: $!";
listen(Server,SOMAXCONN) || die "listen: $!";

logmsg "server started on $NAME";

my $waitedpid;

sub REAPER {
 $waitedpid = wait;
 $SIG{CHLD} = \&REAPER; # maudissez sysV
 logmsg "reaped $waitedpid" . ($? ? " with exit $" : '');
}

$SIG{CHLD} = \&REAPER;

for ($waitedpid = 0;
 accept(Client,Server) || $waitedpid;
 $waitedpid = 0, close Client)
{
 next if $waitedpid;
 logmsg "connection on $NAME";
 spawn sub {
 print "Hello there, it's now ", scalar localtime, "\n";
 exec '/usr/games/fortune' or die "can't exec fortune: $!";
 };
}
}
```

Comme vous le voyez, il est remarquablement similaire au serveur TCP Internet, à tel point que, en fait, nous avons omis plusieurs fonctions identiques - spawn(), logmsg(), ctime(), et REAPER() - qui sont exactement les mêmes que dans l'autre serveur.

Alors pourquoi vouloir utiliser une socket du domaine Unix au lieu d'un tube nommé plus simple ? Parce qu'un tube nommé ne vous procure pas de sessions. Vous ne pouvez pas différencier les données d'un processus de celle d'un autre. Avec la programmation de sockets, vous obtenez une session distincte pour chaque client : c'est pourquoi accept() prend deux arguments.

Par exemple, supposons que vous voulez donner accès à un démon serveur de base de données tournant depuis longtemps à des copains sur le web, mais seulement s'ils passent par une interface CGI. Vous aurez un programme CGI petit et simple qui fera toutes les vérifications et les connexions que vous voudrez, puis agira comme un client du domaine Unix et se connectera à votre serveur privé.

## 44.6 Clients TCP avec IO::Socket

Pour ceux qui préfèrent une interface de plus haut niveau pour la programmation des sockets, le module IO::Socket fournit une approche orientée objet. IO::Socket fait partie de la distribution standard de Perl à partir de la version 5.004. Si vous exploitez une version précédente de Perl, récupérez juste IO::Socket sur le CPAN, où vous trouverez aussi des modules fournissant des interfaces simples pour les systèmes suivants : DNS, FTP, Ident (RFC 931), NIS et NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLeay, Telnet, et Time - pour n'en citer que quelques-uns.

### 44.6.1 Un Client Simple

Voici un client qui crée une connexion TCP avec le service "daytime" sur le port 13 de l'hôte appelé "localhost" et affiche tout ce que le serveur veut bien fournir.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
 Proto => "tcp",
 PeerAddr => "localhost",
 PeerPort => "daytime(13)",
)
 or die "cannot connect to daytime port at localhost";
while (<$remote>) { print }
```

Lorsque vous lancez ce programme, vous devriez obtenir quelque chose qui ressemble à ceci :

```
Wed May 14 08:40:46 MDT 1997
```

Voici ce que signifient les paramètres du constructeur `new` :

#### Proto

C'est le protocole à utiliser. Dans ce cas, le handle de socket retourné sera connecté à une socket TCP, car nous voulons une connexion orienté flux (stream, NDT), c'est-à-dire une connexion qui se comporte assez comme un bon vieux fichier. Toutes les sockets ne sont pas de ce type. Par exemple, le protocole UDP peut être utilisé pour créer une socket datagramme, pour transmettre des messages.

#### PeerAddr

C'est le nom ou l'adresse Internet de l'hôte distant sur lequel le serveur tourne. Nous aurions pu spécifier un nom plus long comme "www.perl.com", ou une adresse comme "204.148.40.9". Pour les besoins de la démonstration, nous avons utilisé le nom d'hôte spécial "localhost", qui doit toujours désigner la machine sur laquelle le programme tourne. L'adresse Internet correspondant à localhost est "127.1", si vous préférez l'utiliser.

#### PeerPort

C'est le nom du service ou le numéro de port auquel nous aimerions nous connecter. Nous aurions pu nous contenter d'utiliser juste "daytime" sur les systèmes ayant un fichier de services système bien configuré, [FOOTNOTE: Le fichier de services système est dans */etc/services* sous Unix] mais juste au cas où, nous avons spécifié le numéro de port (13) entre parenthèses. La simple utilisation du numéro aurait aussi fonctionné, mais les numéros constants rendent nerveux les programmeurs soigneux.

Vous avez remarqué comment la valeur de retour du constructeur `new` est utilisée comme descripteur de fichier dans la boucle `while` ? C'est ce qu'on appelle un descripteur de fichier indirect, une variable scalaire contenant un descripteur de fichier. Vous pouvez l'utiliser de la même façon qu'un descripteur normal. Par exemple, vous pouvez y lire une ligne de la manière suivante :

```
$line = <$handle>;
```

toutes les lignes restantes de cette façon :

```
@lines = <$handle>;
```

et y écrire une ligne ainsi :

```
print $handle "some data\n";
```

### 44.6.2 Un Client Webget

Voici un client simple qui prend en paramètre un hôte distant, puis une liste de documents à récupérer sur cet hôte. C'est un client plus intéressant que le précédent car il commence par envoyer quelque chose avant de récupérer la réponse du serveur,

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host document ..." }
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
foreach $document (@ARGV) {
 $remote = IO::Socket::INET->new(Proto => "tcp",
 PeerAddr => $host,
 PeerPort => "http(80)",
);
 unless ($remote) { die "cannot connect to http daemon on $host" }
 $remote->autoflush(1);
 print $remote "GET $document HTTP/1.0" . $BLANK;
 while (<$remote>) { print }
 close $remote;
}
}
```

le serveur web gérant le service "http", qui est supposé être à son numéro de port standard, le 80. Si le serveur auquel vous essayez de vous connecter est à un numéro de port différent (comme 1080 ou 8080), vous devez le spécifier par le paramètre nommé `PeerPort => 8080`. La méthode `autoflush` est utilisée sur la socket car autrement le système placerait toutes les sorties que nous y envoyons dans un buffer (si vous êtes sur un Mac, vous devrez aussi remplacer tous les "\n" dans votre code qui envoie des données sur le réseau par des "\015\012".)

Se connecter au serveur est seulement la première partie du travail : une fois que vous avez la connexion, vous devez utiliser le langage du serveur. Chaque serveur sur le réseau à son propre petit langage de commande, qu'il attend en entrée. La chaîne que nous envoyons au serveur et qui commence par "GET" est dans la syntaxe HTTP. Dans ce cas, nous demandons simplement chaque document spécifié. Oui, nous créons vraiment une nouvelle connexion pour chaque document, même s'ils se trouvent tous sur le même hôte. Vous avez toujours parlé HTTP de cette façon. Les versions les plus récentes des clients web peuvent demander au serveur distant de laisser la connexion ouverte un petit moment, mais le serveur n'est pas tenu d'honorer une telle requête.

Voici un exemple d'exécution de ce programme, que nous appellerons *webget* :

```
% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found on this server.<P>
</BODY>
```

Ok, ce n'est donc pas très intéressant, car il n'a pas trouvé ce document en particulier. Mais une longue réponse n'aurait pas tenu sur cette page.

Pour avoir une version un peu plus développée de ce programme, vous devriez jeter un oeil sur le programme *lwp-request* inclus dans les modules LWP du CPAN.

### 44.6.3 Client Interactif avec IO::Socket

Bien, ceci est parfait si vous voulez envoyer une commande et obtenir une réponse, mais pourquoi ne pas mettre en place quelque chose de pleinement interactif, un peu à la façon dont *telnet* fonctionne ? Ainsi vous pouvez taper une ligne, obtenir la réponse, taper une autre ligne, avoir la réponse, etc.



Ce client est plus compliqué que les deux que nous avons écrits jusqu'à maintenant, mais si vous êtes sur un système qui supporte le puissant appel `fork`, la solution n'est pas si rude que ça. Une fois que vous avez obtenu la connexion au service avec lequel vous désirez discuter, appelez `fork` pour cloner votre processus. Chacun de ces deux processus identiques à un travail très simple à réaliser : le parent copie tout ce qui provient de la socket sur la sortie standard, pendant que le fils copie simultanément sur la socket tout ce qui vient de l'entrée standard. Accomplir la même chose en utilisant un seul processus serait *bien plus* difficile, car il est plus facile de coder deux processus qui font une seule chose, qu'un seul processus qui en fait deux (Ce principe de simplicité est une pierre angulaire de la philosophie Unix, ainsi que du bon génie logiciel, c'est probablement pour cela qu'il s'est étendu à d'autres systèmes).

Voici le code:

```
#!/usr/bin/perl -w
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port" }
($host, $port) = @ARGV;

cree une connexion tcp a l'hote et sur le port specifie
$handle = IO::Socket::INET->new(Proto => "tcp",
 PeerAddr => $host,
 PeerPort => $port)
 or die "can't connect to port $port on $host: $!";

$handle->autoflush(1); # pour que la sortie y aille tout de suite
print STDERR "[Connected to $host:$port]\n";

coupe le programme en deux processus, jumeaux identiques
die "can't fork: $!" unless defined($kidpid = fork());

le bloc if{} n'est traverse que dans le processus pere
if ($kidpid) {
 # copie la socket sur la sortie standard
 while (defined ($line = <$handle>)) {
 print STDOUT $line;
 }
 kill("TERM", $kidpid); # envoie SIGTERM au fils
}
le bloc else{} n'est traverse que dans le fils
else {
 # copie l'entree standard sur la socket
 while (defined ($line = <STDIN>)) {
 print $handle $line;
 }
}
}
```

La fonction `kill` dans le bloc `if` du père est là pour envoyer un signal à notre processus fils (qui est dans le bloc `else`) dès que le serveur distant a fermé la connexion de son côté.

Si le serveur distant envoie les données un octet à la fois, et que vous avez besoin de ces données immédiatement sans devoir attendre une fin de ligne (qui peut très bien ne jamais arriver), vous pourriez remplacer la boucle `while` dans le père par ce qui suit :

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
 print STDOUT $byte;
}
```

Faire un appel système pour chaque octet que vous voulez lire n'est pas très efficace (c'est le moins qu'on puisse dire) mais c'est le plus simple à expliquer et cela fonctionne raisonnablement bien.

## 44.7 Serveurs TCP avec IO::Socket

Comme toujours, mettre en place un serveur est un peu plus exigeant que de faire tourner un client. Le modèle est que le serveur crée un type de socket spécial qui ne fait rien à part écouter un port particulier pour attendre l'arrivée d'une connexion. Il le fait en appelant la méthode `IO::Socket::INET->new()` avec des arguments légèrement différents de ceux du client.

### Proto

C'est le protocole à utiliser. Comme pour nos clients, nous spécifierons toujours "tcp" ici.

### LocalPort

Nous spécifions un port local dans l'argument `LocalPort`, ce que nous n'avons pas fait pour le client. C'est le nom du service ou le numéro du port dont vous voulez être le serveur (sous Unix, les ports en dessous de 1024 sont réservés au superutilisateur). Dans notre exemple, nous utiliserons le port 9000, mais vous pouvez utiliser n'importe quel port non utilisé couramment sur votre système. Si vous essayez d'en utiliser un qui soit déjà exploité, vous obtiendrez le message "Address already in use". Sous Unix, la commande `netstat -a` vous montrera quels services ont des serveurs en cours.

### Listen

le paramètre `Listen` détermine le nombre maximal de connexion en cours que nous pouvons accepter avant de commencer à rejeter les clients arrivants. Pensez-y comme à une file d'attente pour vos appels téléphoniques. Le module `Socket` de bas niveau a un symbole spécial pour le maximum du système, qui est `SOMAXCONN`.

### Reuse

Le paramètre `Reuse` est nécessaire pour que nous puissions redémarrer notre serveur manuellement sans devoir attendre quelques minutes que les tampons du système soient vidés.

Une fois que la socket générique du serveur a été créée en utilisant les paramètres listés ci-dessus, le serveur attend qu'un nouveau client s'y connecte. Le serveur se bloque dans la méthode `accept`, qui devient finalement une connexion bidirectionnelle avec le client distant (faites un autoflush sur ce handle pour éviter toute mise en tampon).

Pour être plus gentil avec les utilisateurs, notre serveur leur offre un prompt pour qu'ils entrent leurs commandes. La plupart des serveurs ne font pas cela. À cause du prompt sans caractère de nouvelle ligne, vous devrez utiliser la variante `sysread` du client interactif ci-dessus.

Ce serveur accepte cinq commandes différentes, renvoyant la sortie au client. Notez que contrairement à la plupart des serveurs en réseau, celui-ci ne sait gérer qu'un seul client à la fois. Les serveurs multithreads sont traités dans le chapitre 6 du Chameau ("Programmation en Perl", NDT, avec un dromadaire sur la couverture).

Voici le code.

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent; # pour la version 00 de gethostbyaddr

$PORT = 9000; # choisir quelque chose qui n'est pas utilise

$server = IO::Socket::INET->new(Proto => 'tcp',
 LocalPort => $PORT,
 Listen => SOMAXCONN,
 Reuse => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";

while ($client = $server->accept()) {
 $client->autoflush(1);
 print $client "Welcome to $0; type help for command list.\n";
 $hostinfo = gethostbyaddr($client->peeraddr);
 printf "[Connect from %s]\n", $hostinfo->name || $client->peerhost;
 print $client "Command? ";
 while (<$client>) {
 next unless /\S/; # ligne vide
 if (/quit|exit/i) { last; }
 elsif (/date|time/i) { printf $client "%s\n", scalar localtime; }
 }
}
```

```

 elsif (/who/i) { print $client 'who 2>&1'; }
 elsif (/cookie/i) { print $client '/usr/games/fortune 2>&1'; }
 elsif (/motd/i) { print $client 'cat /etc/motd 2>&1'; }
 else {
 print $client "Commands: quit date who cookie motd\n";
 }
} continue {
 print $client "Command? ";
}
close $client;
}

```

## 44.8 UDP : Transfert de Message

Un autre type de configuration client-server n'utilise pas de connexions, mais des messages. Les communications UDP nécessitent moins de ressources mais sont aussi moins fiables, car elles ne promettent pas du tout que les messages vont arriver, sans parler de leur ordre ou de l'état dans lequel ils seront. Toutefois, l'UDP offre certains avantages sur TCP, à commencer par la capacité à faire des "broadcasts" ou des "multicasts" à destination de tout un tas d'hôtes d'un seul coup (habituellement sur votre sous-réseau local). Si vous êtes très inquiet sur la fiabilité et commencez à mettre en place des vérifications dans votre système de messages, alors vous devriez probablement juste utiliser TCP pour commencer.

Voici un programme UDP similaire au client TCP Internet donné précédemment. Toutefois, au lieu de contacter un hôte à la fois, la version UDP en contactera de nombreux de façon asynchrone en simulant un multicast puis en utilisant `select()` pour attendre une activité sur les E/S pendant dix secondes. Pour faire quelque chose de similaire en TCP, vous seriez obligé d'utiliser un handle de socket différent pour chaque hôte.

```

#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ($count, $hisiaddr, $hispaddr, $histime,
 $host, $iaddr, $paddr, $port, $proto,
 $rin, $rout, $rtime, $SECS_of_70_YEARS);

$SECS_of_70_YEARS = 2208988800;

$iaddr = gethostbyname(hostname());
$proto = getprotobyname('udp');
$port = getservbyname('time', 'udp');
$paddr = sockaddr_in(0, $iaddr); # 0 means let kernel pick

socket(SOCKET, PF_INET, SOCK_DGRAM, $proto) || die "socket: $!";
bind(SOCKET, $paddr) || die "bind: $!";

$| = 1;
printf "%-12s %8s %s\n", "localhost", 0, scalar localtime time;
$count = 0;
for $host (@ARGV) {
 $count++;
 $hisiaddr = inet_aton($host) || die "unknown host";
 $hispaddr = sockaddr_in($port, $hisiaddr);
 defined(send(SOCKET, 0, 0, $hispaddr)) || die "send $host: $!";
}

$rin = '';
vec($rin, fileno(SOCKET), 1) = 1;

```

```
timeout after 10.0 seconds
while ($count && select($rout = $rin, undef, undef, 10.0)) {
 $rtime = '';
 ($hisppaddr = recv(SOCKET, $rtime, 4, 0)) || die "recv: $!";
 ($port, $hisiaddr) = sockaddr_in($hisppaddr);
 $host = gethostbyaddr($hisiaddr, AF_INET);
 $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
 printf "%-12s ", $host;
 printf "%8d %s\n", $histime - time, scalar localtime($histime);
 $count--;
}

```

## 44.9 CIP du SysV

Même si la CIP du System V n'est pas aussi largement utilisée que les sockets, elle a des usages intéressants. Vous ne pouvez pas, toutefois, utiliser efficacement la CIP du Système V ou le `mmap()` de Berkeley pour obtenir de la mémoire partagée afin de partager une variable entre plusieurs processus. C'est parce que Perl réallouerait votre chaîne alors que vous ne le voulez pas.

Voici un petit exemple montrant l'utilisation de la mémoire partagée.

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRWXU);

$size = 2000;
$id = shmget(IPC_PRIVATE, $size, S_IRWXU) || die "$!";
print "shm key $id\n";

$message = "Message #1";
shmwrite($id, $message, 0, 60) || die "$!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60) || die "$!";
print "read : '$buff'\n";

le tampon de shmread est cadre a droite par des caracteres nuls.
substr($buff, index($buff, "\0")) = '';
print "un" unless $buff eq $message;
print "swell\n";

print "deleting shm $key\n";
shmctl($id, IPC_RMID, 0) || die "$!";

```

Voici un exemple de sémaphore :

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT) || die "$!";
print "shm key $id\n";

```

Mettez ce code dans un fichier séparé pour être exécuté dans plus d'un processus. Appelez le fichier *take* :

```
creation d'un semaphore

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0 , 0);
die if !defined($id);

$semnum = 0;
$semflag = 0;

```

```

semaphore 'take'
on attend que le semaphore soit à zero
$semop = 0;
$opstring1 = pack("s!s!s!", $semnum, $semop, $semflag);

on incremente le compteur du semaphore
$semop = 1;
$opstring2 = pack("s!s!s!", $semnum, $semop, $semflag);
$opstring = $opstring1 . $opstring2;

semop($id,$opstring) || die "$!";

```

Mettez ce code dans un fichier séparé pour être exécuté dans plus d'un processus. Appelez ce fichier *give* :

```

'give' le semaphore
faites tourner ceci dans le processus originel, et vous verrez
que le second processus continue

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
die if !defined($id);

$semnum = 0;
$semflag = 0;

Decremente le compteur du semaphore
$semop = -1;
$opstring = pack("s!s!s!", $semnum, $semop, $semflag);

semop($id,$opstring) || die "$!";

```

Le code de CIP SysV ci-dessus fut écrit il y a longtemps, et il est définitivement déglingué. Pour un look plus moderne, voir le module IPC::SysV qui est inclus dans Perl à partir de la version 5.005.

Un petit exemple démontrant les files de messages SysV :

```

use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);

my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRWXU);

my $sent = "message";
my $type = 1234;
my $rcvd;
my $type_rcvd;

if (defined $id) {
 if (msgsnd($id, pack("l! a*", $type_sent, $sent), 0)) {
 if (msgrcv($id, $rcvd, 60, 0, 0)) {
 ($type_rcvd, $rcvd) = unpack("l! a*", $rcvd);
 if ($rcvd eq $sent) {
 print "okay\n";
 } else {
 print "not okay\n";
 }
 } else {
 die "# msgrcv failed\n";
 }
 } else {
 die "# msgsnd failed\n";
 }
 msgctl($id, IPC_RMID, 0) || die "# msgctl failed: $!\n";
} else {
 die "# msgget failed\n";
}

```

## 44.10 NOTES

La plupart de ces routines retournent silencieusement mais poliment `undef` lorsqu'elles échouent, au lieu de mourir sur-le-champ à cause d'une exception non piégée (en vérité, certaines des nouvelles fonctions de conversion *Socket* font un `croak()` sur les arguments mal définis). Il est par conséquent essentiel de vérifier les valeurs de retour de ces fonctions. Débutez toujours vos programmes utilisant des sockets de cette façon pour obtenir un succès optimal, et n'oubliez pas d'ajouter dans les serveurs l'option **-T** de vérification de pollution dans la ligne `#!` :

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

## 44.11 BUGS

Toutes ces routines créent des problèmes de portabilité spécifiques à chaque système. Comme il est noté par ailleurs, Perl est à la merci de vos bibliothèques C pour la plus grande partie de comportement au niveau système. Il est probablement plus sûr de considérer comme déficiente la sémantique des signaux de SysV et de s'en tenir à de simples opérations TCP et UDP sur les sockets ; e.g., n'essayez pas de passer des descripteurs de fichiers ouverts par une socket datagramme UDP locale si vous voulez que votre code ait une chance d'être portable.

Comme il l'a été mentionné dans la section sur les signaux, puisque peu de fournisseurs offrent des bibliothèques C réentrantes de façon sûre, le programmeur prudent fera peut de choses dans un handler à part modifier la valeur d'une variable numérique préexistante ; ou, s'il est bloqué dans un appel système lent (redémarrage), il utilisera `die()` pour lever une exception et sortir par un `longjmp(3)`. En fait, même ceci peut dans certains cas provoquer un `coredump`. Il est probablement meilleur d'éviter les signaux sauf lorsqu'ils sont absolument inévitables. Ce problème sera réglé dans une future version de Perl.

## 44.12 VOIR AUSSI

La programmation d'applications en réseau est bien plus vaste que ceci, mais cela devrait vous permettre de débiter.

Pour les programmeurs intrépides, le livre indispensable est *Unix Network Programming* par W. Richard Stevens (publié chez Addison-Wesley). Notez que la plupart des livres sur le réseau s'adressent au programmeur C ; la traduction en Perl est laissée en exercice pour le lecteur.

La page de manuel `IO::Socket(3)` décrit la bibliothèque objet, et la page `Socket(3)` l'interface de bas niveau vers les sockets. Au-delà des fonctions évidentes dans *perlfunc*, vous devriez aussi jeter un oeil sur le fichier *modules* de votre miroir CPAN le plus proche (Voir *perlmodlib* ou mieux encore, la *Perl FAQ*, pour une description de ce qu'est le CPAN et pour savoir où le trouver).

La section 5 du fichier *modules* est consacrée au "Networking, Device Control (modems), and Interprocess Communication", et contient de nombreux modules en vrac, de nombreux modules concernant le réseau, les opérations de Chat et d'Expect, la programmation CGI, le DCE, le FTP, l'IPC, NNTP, les Proxy, Ptty, les RPC, le SNMP, le SMTP, Telnet, les Threads, et ToolTalk - pour n'en citer que quelques-uns.

## 44.13 AUTEUR

Tom Christiansen, avec des vestiges occasionnels de la version originale de Larry Wall et des suggestions des porteurs de Perl.

## 44.14 TRADUCTION

### 44.14.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

**44.14.2 Traducteur**

Roland Trique <roland.trique@free.fr>

Conseils : Guy Decoux <decoux@moulon.inra.fr>, Guillaume van der Rest <vanderrest@magnet.fsu.edu>

**44.14.3 Relecture**

Guy Decoux <decoux@moulon.inra.fr>, Guillaume van der Rest <vanderrest@magnet.fsu.edu>

# Chapitre 45

## perlnumber

Sémantique des nombres et opérations numériques en Perl

### 45.1 SYNOPSIS

```
$n = 1234; # entier en décimal
$n = 0b1110011; # entier en binaire
$n = 01234; # entier en octal
$n = 0x1234; # entier en hexadécimal
$n = 12.34e-56; # notation exponentielle
$n = "-12.34e56"; # nombre spécifié sous forme de chaîne
$n = "1234"; # nombre spécifié sous forme de chaîne
```

### 45.2 DESCRIPTION

Ce document décrit comment Perl manipule les valeurs numériques en interne.

Les possibilités de surcharge d'opérateurs en Perl sont complètement ignorées ici. La surcharge d'opérateurs permet à l'utilisateur d'adapter le comportement des nombres pour obtenir, par exemple, des opérations sur des entiers ou des flottants en précision arbitraire ou des opérations sur des nombres "exotiques" telles l'arithmétique modulaire ou l'arithmétique p-adique et ainsi de suite. Voir *overload* pour plus de détails.

### 45.3 Stockage des nombres

Perl, en interne, peut représenter un nombre de trois manières différentes : comme un entier natif, comme un flottant natif et comme une chaîne de caractères en décimal. Les chaînes de caractères en décimal peuvent contenir un exposant, comme dans "12.34e-56". *Natif* signifie ici "un format reconnu par le compilateur C qui a été utilisé pour construire perl".

Le terme "natif" n'a pas autant d'implications lorsqu'on parle d'entiers natifs que quand on parle de flottants natifs. La seule implication du terme "natif" appliqué aux entiers est que les limites maximales et minimales acceptées pour une quantité entière sont très proches d'une puissance de 2. En revanche, les flottants "natifs" ont plus de restrictions : ils ne peuvent représenter que des nombres qui ont une représentation relativement "courte" lorsqu'ils sont convertis en base 2. Par exemple, 0.9 ne peut pas être représenté par un flottant natif puisque son écriture en base 2 est infinie :

```
0.1110011001100...
```

avec la séquence 1100 qui se répète indéfiniment ensuite. En plus de la limitation précédente, la partie exposant du nombre est, elle aussi, limitée lorsqu'elle est représentée par un flottant. Sur la plupart des matériels, les valeurs flottantes peuvent stocker des nombres contenant au plus 53 chiffres binaires avec une partie exposant comprise entre -1024 et 1024. Une fois ramenées en décimale, ces limites donnent environ 16 chiffres significatifs et une puissance de dix comprises entre -304 et 304. Une conséquence de tout cela est donc que, sur de telles architectures, Perl ne peut pas stocker dans un flottant un nombre comme 12345678901234567 sans perdre d'information.



De manière similaire, les chaînes de caractères en décimal ne peuvent représenter que des nombres dont la représentation décimale est finie. Comme ce sont des chaînes, et donc de longueur quelconque, il n'y a pratiquement pas de limite pour l'exposant ou le nombre de décimales de tels nombres. (Mais n'oubliez pas que nous ne parlons que des règles de *stockage* de ces nombres. Que vous puissiez effectivement stocker des "grands" nombres n'implique pas que vous pourrez réaliser des *opérations* sur ces nombres utilisant tous les chiffres disponibles. Voir Opérateurs et conversions numériques (§45.4) pour plus de détails.)

Dans les faits, les nombres stockés au format natif entier peuvent l'être sous leur forme avec ou sans signe. Donc, en Perl, les limites pour les entiers stockés au format natif sont typiquement  $-2^{31}..2^{32}-1$ , valeurs à adapter dans le cas des entiers sur 64 bits. Encore une fois, cela ne signifie pas que Perl peut faire des opérations uniquement sur des entiers dans cet intervalle : il est possible de stocker bien plus d'entiers en utilisant le format flottant.

En résumé, les valeurs numériques que Perl peut stocker sont celles qui ont une représentation décimale finie ou une "courte" représentation binaire.

## 45.4 Opérateurs et conversions numériques

Perl peut stocker un nombre dans n'importe lequel des trois formats mentionnés ci-dessus mais, typiquement, la plupart des opérateurs ne comprennent qu'un seul de ces formats. Lorsqu'une valeur numérique est passée comme argument à l'un de ces opérateurs, elle est convertie vers le format compris par l'opérateur.

Six conversions sont donc possibles :

```
entier natif --> flottant natif (*)
entier natif --> chaîne décimale
flottant natif --> entier natif (*)
flottant natif --> chaîne décimale (*)
chaîne décimale --> entier natif
chaîne décimale --> flottant natif (*)
```

Ces conversions suivent les règles générales suivantes :

- Si le nombre d'origine peut être représenté sous la forme visée, cette représentation est utilisée.
- Si le nombre d'origine est en dehors des limites représentables par la forme visée, la représentation choisie est celle de la limite la plus proche. (*perte d'information*)
- Si le nombre d'origine est compris entre deux nombres représentables par la forme visée, la représentation de l'un de ces deux nombres est utilisée. (*perte d'information*)
- Lors de la conversion `flottant natif -> entier natif`, la partie entière du résultat est toujours inférieure ou égale à la partie entière de la valeur d'origine. ("*arrondi vers zéro*")
- Si la conversion `chaîne décimale -> entier natif` ne peut pas être réalisée sans perte d'information, le résultat est compatible avec la suite de conversions `chaîne décimale -> flottant natif -> entier natif`. En particulier, cela donne une chance à un nombre tel que `"0.999999999999999999"` d'être arrondi à 1.

**RESTRICTION** : les conversions ci-dessus marquées d'une \* sont en partie effectués par le compilateur C. Donc, des bogues ou des particularités du compilateur utilisé peuvent parfois amener au non respect de certaines des règles ci-dessus.

## 45.5 Un avant-goût des opérations numériques en Perl

Les opérations Perl qui prennent un argument numérique traitent cet argument selon l'une des quatre manières suivantes : soit elles le transforment vers l'un des trois formats entier, flottant ou chaîne décimale, soit elles se comportent différemment selon le format d'origine de l'argument. La conversion de la valeur numérique vers un format particulier ne change pas le nombre stocké dans la valeur.

Tous les opérateurs qui nécessitent un argument au format entier traitent leur argument en arithmétique modulaire, par exemple `mod 2**32` sur une architecture 32-bits. `spintf "%u"`, `-1` fournira donc le même résultat que `spintf "%u"`, `~0`.

### Les opérateurs arithmétiques

Les opérateurs binaires `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `>`, `<`, `>=` et `<=` ainsi que les opérateurs unaires `-`, `abs` et `-` essayeront de convertir leurs arguments en entiers. Si les deux conversions sont possibles sans perte de précision et si l'opération peut être effectuée sans perte de précision alors un résultat entier sera produit. Sinon les arguments sont convertis en flottant et le résultat sera un flottant. Les conversions utilisant une sorte de cache (comme décrit ci-dessous), les conversions vers des entiers ne perdront pas la partie décimale des nombres flottants.

**++**

++ se comportent comme les opérateurs ci-dessus sauf si son argument est une chaîne de caractères qui est reconnue par l'expression rationnelle `/^[a-zA-Z]*[0-9]*\z/`. Dans ce dernier cas, c'est l'incrément de chaîne décrite dans *perlop* qui est utilisée.

#### **Les opérateurs arithmétiques lorsque `use integer` est actif**

Si `use integer;` est actif, la quasi totalité des opérateurs listés ci-dessus convertissent leur(s) argument(s) au format entier et retourne un résultat entier. Les exceptions sont `abs`, `++` et `-` qui ne changent pas leur comportement.

#### **Les autres opérateurs mathématiques**

Les opérateurs tels que `**`, `sin` et `exp` convertissent leurs arguments vers le format flottant.

#### **Les opérateurs bit-à-bit**

Les arguments sont convertis au format entier si ce ne sont pas des chaînes de caractères.

#### **Les opérateurs bit-à-bit lorsque `use integer` est actif**

Les arguments sont convertis au format entier. De plus les opérations internes de décalage utilisent des entiers signés au lieu des non signés par défaut.

#### **Les opérations qui attendent un entier**

L'argument est converti au format entier. Ceci s'applique, par exemple, au troisième et au quatrième argument de `sysread`.

#### **Les opérations qui attendent une chaîne**

L'argument est converti au format chaîne décimale. Par exemple, cela s'applique à `<printf "%s", $value>`.

Bien que la conversion d'un argument vers un format particulier ne change pas le nombre stocké, Perl se souvient du résultat de ces conversions. En particulier, bien que la première conversion puisse prendre du temps, des opérations répétées n'auront plus à refaire cette conversion.

## **45.6 AUTEUR**

Ilya Zakharevich [ilya@math.ohio-state.edu](mailto:ilya@math.ohio-state.edu)

Quelques adaptations par Gurusamy Sarathy [<gsar@ActiveState.com>](mailto:gsar@ActiveState.com)

Mise à jour pour 5.8.0 par Nicholas Clark [<nick@ccl4.org>](mailto:nick@ccl4.org)

## **45.7 VOIR AUSSI**

*overload* et *perlop*.

## **45.8 TRADUCTION**

### **45.8.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **45.8.2 Traducteur**

Paul Gaborit ([Paul.Gaborit@enstimac.fr](mailto:Paul.Gaborit@enstimac.fr)).

### **45.8.3 Relecture**

Personne pour l'instant.

# Chapitre 46

## perlthrtut

Tutoriel sur les threads en Perl

### 46.1 DESCRIPTION

Ce tutoriel décrit les threads à la nouvelle mode introduites dans Perl 5.6.0, appelés threads interpréteur, ou **ithreads** pour faire court. Dans ce modèle, chaque thread exécute son propre interpréteur Perl, et tout partage de données entre les threads doit être explicite.

Il y a une autre, plus vieille mode de gestion des threads en Perl, appelée le modèle 5.005 ; évidemment, il s'applique aux versions 5.005 de Perl. Ce vieux modèle a des problèmes connus, est obsolète, et sera probablement retiré autour de la version 5.10. Vous êtes fortement encouragé à faire migrer aussi vite que possible tout code utilisant le modèle 5.005 vers le nouveau modèle.

Vous pouvez savoir quel modèle de threads vous avez (ou savoir si vous n'en avez aucun) en exécutant `perl -V` et en examinant la section `Platform`. Si vous avez `useithreads=define`, vous avez les `ithreads` ; si vous avez `use5005threads=define`, vous avez les threads 5.005. Si vous n'avez aucun des deux, votre perl ne contient pas de support pour les threads. Si vous avez les deux, vous avez un problème.

L'accès aux threads 5.005 se faisait à travers la classe *Threads*, alors que celui aux `ithreads` se fait grâce à la classe *threads*. Notez le `t` majuscule et minuscule.

### 46.2 État courant

Le code du modèle `ithreads` existe depuis Perl 5.6.0, et est considéré stable. L'interface utilisateur aux `ithreads` (la classe *threads*) est apparue dans la version 5.8.0, et aujourd'hui on peut la considérer comme stable, bien qu'il faille la traiter avec précautions comme tout ce qui est nouveau.

### 46.3 Qu'est-ce que c'est qu'un thread, d'abord ?

Un thread (*processus léger* en bon Français) est un flux de contrôle à travers un programme, muni d'un point d'exécution unique.

Ça ressemble vraiment à un processus, non ? Eh bien, c'est normal. Les threads sont un des composants d'un processus. Chaque processus a au moins un thread, et d'ailleurs jusqu'à aujourd'hui, chaque processus exécutant Perl a eu un seul thread. Avec Perl 5.8, cependant, vous pouvez créer des threads supplémentaires. Nous allons voir comment, quand, et pourquoi.

### 46.4 Modèles de programmes utilisant les threads

Il y a trois façon de structurer un programme utilisant des threads. Le choix du modèle dépend de ce que vous voulez que votre programme fasse. Pour de nombreux programmes non triviaux utilisant les threads, vous allez devoir choisir des modèles différents pour les différentes parties de votre programme.

### 46.4.1 Maître/Esclave

Le modèle maître/esclave met en scène un seul thread "maître", et un thread "esclave" ou plus. Le thread maître rassemble et génère les tâches qui doivent être accomplies, et répartit ces tâches aux threads esclaves.

Ce modèle est courant dans les programmes serveurs et à interface graphique, où un thread principal attend un évènement et le passe aux threads esclaves appropriés, pour qu'ils le traitent. Une fois l'évènement transmis, le thread maître se remet à attendre un autre évènement.

Le thread maître fait assez peu de travail. Même si les tâches ne sont pas nécessairement traitées plus vite qu'avec une autre méthode, celle-ci a tendance à présenter les meilleurs temps de réponse à l'utilisateur.

### 46.4.2 Équipe de travail

Dans le modèle de l'équipe de travail, plusieurs threads traitent de la même façon les différents morceaux de données. Cela reproduit le mode de travail du calcul parallèle classique et des processeurs vectoriels, où une série de processeurs font exactement la même chose à de nombreux morceaux de données.

Ce modèle est particulièrement utile si le système qui exécute le programme peut distribuer les threads sur les différents processeurs. Il peut aussi être utile en lancer de rayons ou dans les moteurs de rendu, quand les threads individuels peuvent renvoyer des résultats intermédiaires pour donner un retour visuel à l'utilisateur.

### 46.4.3 Travail à la chaîne

Le modèle de travail à la chaîne divise une tâche en une suite d'étapes, et consiste à passer les résultats d'une étape au thread qui s'occupe de l'étape suivante. Chaque thread fait une chose à chaque morceau de données et passe ses résultats au thread suivant de la chaîne.

Ce modèle est surtout conseillé si vous avez plusieurs processeurs, de façon à ce que plusieurs threads s'exécutent en parallèle, même si il peut aussi servir dans d'autres contextes. Il permet de garder les tâches élémentaires simples, et autorise certaines parties de la chaîne à bloquer (sur des appels système d'E/S, par exemple) alors que d'autres parties continuent à tourner. Si vous exécutez les différentes parties de la chaîne sur des processeurs différents, cela vous permet aussi de tirer avantage du cache de chaque processeur.

Ce modèle est aussi pratique pour une forme de programmation récursive dans laquelle au lieu de s'appeler elle-même, une fonction crée un autre thread. Les générateurs de nombres premiers ou de Fibonacci profitent bien du modèle de travail à la chaîne. (Une version d'un générateur de nombres premiers est présentée plus bas.)

## 46.5 Implémentations des threads dans le système d'exploitation

Il y a plusieurs façons différentes d'implémenter les threads sur un système donné. Cela dépend de la marque, et, parfois, de la version du système d'exploitation (SE). Souvent la première implémentation est relativement simple, mais celles des versions suivantes du SE sont plus sophistiquées.

Les informations présentées dans cette section sont utiles mais pas indispensables ; vous pouvez donc les sauter si elles vous rebutent.

Il y a trois sortes de threads : les threads en mode utilisateur, les threads du noyau, et les threads multiprocesseurs du noyau.

Les threads en mode utilisateur vivent entièrement à l'intérieur d'un programme et de ses bibliothèques. Dans ce modèle, le SE ne sait rien des threads. Pour ce qu'il en voit, votre processus utilisant les threads n'est qu'un processus comme les autres.

C'est la façon la plus facile d'implémenter les threads, et celle avec laquelle commencent la plupart des SE. Le gros désavantage est que, puisque le SE ne sait rien des threads, si un des thread bloque, ils bloquent tous. Parmi les activités bloquantes courantes, on trouve la plupart des appels système, des E/S, et les choses comme `sleep()`.

Les threads du noyau sont l'étape suivante dans l'évolution des threads. Le SE connaît des threads gérés par le noyau et en tient compte. La principale différence entre un thread du noyau et un thread en mode utilisateur est le blocage. Avec les threads du noyau, un thread peut bloquer sans que les autres threads bloquent. Ce n'est pas le cas avec les threads en mode utilisateur, où le noyau bloque au niveau du processus et pas au niveau du thread.

C'est un grand pas en avant, qui peut donner à un programme utilisant les threads un bonus de performance sur les programmes ne les utilisant pas. Les threads qui bloquent au cours d'E/S, par exemple, ne bloqueront pas les threads qui

font autre chose. Cependant, chaque processus ne peut avoir qu'un thread en train de s'exécuter à un instant donné, quel que soit le nombre de processeurs que le système a.

Puisque le noyau peut interrompre un thread à n'importe quel moment, cela va mettre à nu certaines des hypothèses implicites d'exclusion que vous pouvez faire dans votre programme. Par exemple, quelque chose d'aussi simple que  $\$a = \$a + 2$  peut se comporter de façon imprévisible avec des threads du noyau si  $\$a$  est visible aux autres threads : un autre thread peut en effet changer  $\$a$  entre le moment où  $\$a$  est lue dans la partie droite, et celui où la nouvelle valeur est stockée.

Les threads multiprocesseurs du noyau sont l'étape finale dans l'évolution du support pour les threads. Avec ce modèle, sur une machine à plusieurs processeurs, le SE peut faire tourner simultanément plusieurs threads sur plusieurs processeurs.

Cela peut améliorer sérieusement les performances d'un programme utilisant les threads, puisque plusieurs s'exécuteront en même temps. En retour, cependant, tous les fourbes problèmes de synchronisation qui ne se montraient pas avec les threads du noyau vont surgir pour se venger.

Au-delà des différents niveaux d'implication des SE dans la gestion des threads, il y a différentes façon pour un SE (et pour une implémentation des threads sur un même SE) d'allouer des cycles processeur aux threads.

Les systèmes multitâches coopératifs obligent les threads qui s'exécutent à leur rendre le contrôle dans deux cas. Un thread peut demander explicitement de céder le contrôle. Il le cède aussi s'il fait quelque chose de bloquant, comme des E/S. Dans un système multitâche coopératif, un thread peut priver tous les autres de temps processeur si il le décide.

Les systèmes multitâches préemptifs interrompent les threads à intervalle régulier, et le système décide quelle thread doit être exécuté ensuite. Sur un tel système, un thread ne peut généralement pas monopoliser le processeur.

Sur certains systèmes, des threads coopératifs et préemptifs peuvent tourner simultanément. (Par exemple, Les threads avec des priorités temps-réel se comportent souvent coopérativement, alors que ceux avec des priorités normales se comportent préemptivement.)

## 46.6 De quelle sorte sont les threads de Perl ?

Si vous avez expérimenté d'autres implémentations des threads, vous pourriez avoir l'impression que les choses ne sont pas ce à quoi vous vous attendiez. Avec les threads de Perl, il est très important de garder à l'esprit qu'ils ne sont d'aucune école particulière. Ce ne sont pas des threads POSIX, ni DecThreads, ni des Green Threads de Java, ni des threads Win32. Il y a des similarités, et les concepts généraux sont les mêmes, mais si vous commencez à mettre votre nez dans les détails d'implémentation, vous allez être déçu, ou troublé. Les deux peut-être.

Cela ne veut pas dire que les threads de Perl sont différents de tout ce qui est venu avant. Le modèle de Perl doit beaucoup aux autres modèles, surtout au modèle POSIX. Mais de la même façon que Perl n'est pas C, les threads Perl ne sont pas les threads POSIX. Donc, si vous commencez à chercher des mutexes ou des priorités de threads, prenez du recul et repensez à ce que vous voulez faire, et à comment Perl peut le faire.

Cependant, il est important de se souvenir que les threads de Perl ne peuvent pas faire des choses que n'autorise pas votre système d'exploitation. Si celui-ci bloque un processus entier sur `sleep()`, Perl va bloquer aussi.

Les Threads de Perl Sont Différents.

## 46.7 Modules réentrants (*thread-safe*)

L'ajout des threads a changé l'intérieur de Perl substantiellement. Cela a des implications pour ceux qui écrivent des modules utilisant XS ou des bibliothèques externes. Cependant, dans la mesure où par défaut les données ne sont pas partagées par les threads, les modules Perl ont de grandes chances d'être déjà réentrants, ou peuvent le devenir facilement. Il faut tester ou passer en revue le code des modules non réentrants avant de les utiliser en production.

Tous les modules que vous pouvez utiliser ne sont pas réentrants, et vous devriez toujours supposer qu'un module ne l'est pas, à moins que sa documentation ne dise explicitement le contraire. Cela tient pour les modules qui sont distribués dans la version standard de Perl. Les threads sont une caractéristique nouvelle, et même certains des modules standards ne sont pas réentrants.

Même si un module est réentrant, cela n'implique pas qu'il soit optimisé pour fonctionner avec les threads. Il est possible qu'il puisse être réécrit pour tirer parti des nouvelles fonctionnalités liées aux threads de Perl, pour augmenter les performances dans un environnement utilisant les threads.

Si vous utilisez un module qui se trouve n'être pas réentrant, vous pouvez vous protéger en ne l'utilisant que depuis un seul et unique thread. Si vous avez besoin que de plusieurs threads accèdent à ce module, il vous reste à utiliser des sémaphores et beaucoup de discipline. Les sémaphores sont décrits dans Sémaphores de base (§46.10.5).

Voir aussi Réentrance des bibliothèques système (§46.15).

## 46.8 Bases des threads

Le module standard *threads* fournit les fonctions de base nécessaires à l'écriture de programmes utilisant les threads. Dans les sections suivantes, nous allons en décrire les bases, en vous montrant ce dont vous avez besoin pour faire tourner un programme avec des threads. Après ça, nous passerons en revue certaines fonctionnalités du module *threads* qui rendent la vie avec les threads plus facile.

### 46.8.1 Support de base pour les threads

Le support de Perl pour les threads est une option de compilation - il est activé ou désactivé quand Perl est compilé sur votre machine, et non quand vos propres programmes sont compilés. Si votre perl n'a pas été compilé avec le support pour les threads, toute tentative d'utiliser les threads est vouée à l'échec.

Vos programmes peuvent utiliser le module *Config* pour déterminer si les threads sont activés ou non. Si votre programme ne peut tourner sans eu, vous pouvez écrire quelque chose comme :

```
$Config{useithreads} or die
 "Recompilez Perl avec les threads activés pour faire tourner ce programme.";
```

Si nous devons faire usage d'un module capable d'utiliser les threads ou non, nous pouvons écrire un programme qui les utilise si possible :

```
use Config;
use MonModule;

BEGIN {
 if ($Config{useithreads}) {
 # Nous avons les threads
 require MonModule_avec_threads;
 import MonModule_avec_threads;
 } else {
 require MonModule_sans_threads;
 import MonModule_sans_threads;
 }
}
```

Du code capable de tourner avec et sans les threads est généralement très touffu, aussi est-il préférable d'isoler le code spécifique aux threads dans son propre module. Dans l'exemple ci-dessus, c'est ce que fait *MonModule\_avec\_threads*, qui est seulement importé si nous tournons dans un perl avec les threads activés.

### 46.8.2 Note à propos des exemples

Bien que le support pour les threads soit considéré comme stable, il reste un certain nombre de rugosités qui pourraient vous gêner si vous essayez les exemples ci-dessous. Dans une situation réelle, il faudrait prendre garde que tous les threads aient fini avant que le programme ne se termine. Cette précaution n'a **pas** été prise dans nos exemples, par souci de simplicité. Tels quels, ils vont produire des messages d'erreur, généralement à cause de threads qui tournent encore quand le programme se termine. Ne vous en effrayez pas. Les versions futures de Perl régleront peut-être ce problème.

### 46.8.3 Créer des threads

Le module *threads* fournit les outils nécessaires à la création de threads. Comme n'importe quel autre module, vous devez dire à Perl que vous voulez l'utiliser ; `use threads` importe tous les éléments dont vous avez besoin pour créer des threads.

La façon la plus simple de créer un thread utilise `new()` :

```
use threads;

$thr = threads->new(\&sub1);
```

```
sub sub1 {
 print "Dans le thread\n";
}
```

La méthode `new()` prend une référence à une fonction et crée un nouveau thread, qui commence à s'exécuter dans ladite fonction. Le contrôle est alors à la fois dans la fonction et dans l'appelant.

Si nécessaire, votre programme peut passer des paramètres à la fonction au démarrage du thread. Il suffit d'inclure la liste des paramètres dans l'appel à `threads::new`, comme ceci :

```
use threads;

$Param3 = "toto";
$thr = threads->new(\&sub1, "Param 1", "Param 2", $Param3);
$thr = threads->new(\&sub1, @ListeDeParametres);
$thr = threads->new(\&sub1, qw(Param1 Param2 Param3));

sub sub1 {
 my @Parametres = @_;
 print "Dans le thread\n";
 print "Reçu les paramètres >", join("<>", @Parametres), "<\n";
}
```

Cet exemple illustre une autre caractéristique des threads : vous pouvez générer plusieurs threads en utilisant la même fonction. Chaque thread exécute la même fonction, mais avec un environnement séparé et des arguments potentiellement différents.

`create()` est un synonyme de `new()`.

#### 46.8.4 Rendre le contrôle

On veut parfois qu'un thread rende explicitement le contrôle du processeur à un autre thread. Par exemple, votre système de gestion des threads pourrait ne pas supporter le multitâche préemptif, ou vous pourriez être en train de faire quelque chose de très lourd en calculs, et vouloir être sûr que le thread de l'interface utilisateur soit appelé assez souvent. Et il y a des cas où l'on veut juste qu'un thread lâche le contrôle du processeur.

Le système de threads de Perl fournit la fonction `yield()`, qui permet de faire ceci. `yield()` a une utilisation assez évidente :

```
use threads;

sub boucle {
 my $thread = shift;
 my $toto = 50;
 while($toto-- > 0) { print "dans le thread $thread\n" }
 threads->yield;
 $toto = 50;
 while($toto-- > 0) { print "dans le thread $thread\n" }
}

my $thread1 = threads->new(\&boucle, 'premier');
my $thread2 = threads->new(\&boucle, 'deuxième');
my $thread3 = threads->new(\&boucle, 'troisième');
```

Il est important de garder à l'esprit que `yield()` n'est qu'une suggestion de rendre le contrôle du processeur, et qu'il dépend de votre matériel, SE et bibliothèques de threads que cela arrive réellement. Par conséquent, il ne faut pas espérer imposer l'ordonnancement des threads avec des appels à `yield()`. Cela peut marcher sur votre plateforme, mais cela ne marchera pas sur toutes.

### 46.8.5 Attendre qu'un thread termine

Puisque les threads sont aussi des fonctions, elles peuvent renvoyer des valeurs. Pour attendre qu'un thread termine et utiliser les valeurs qu'il peut retourner, vous pouvez vous servir de la méthode `join()` :

```
use threads;

$thr = threads->new(\&sub1);

@DonneesRenvoyees = $thr->join;
print "Le thread a renvoyé @DonneesRenvoyees";

sub sub1 { return "Cinquante-six", "toto", 2; }
```

Dans cet exemple, la méthode `join()` retourne dès que le thread se termine. En plus d'attendre qu'un thread termine et de rassembler les valeurs qu'il peut retourner, `join()` effectue aussi le nettoyage nécessaire. Ce nettoyage peut prendre des ressources importantes, particulièrement pour les programmes qui tournent longtemps et génèrent de nombreux threads. Si vous ne voulez pas récupérer les valeurs de retour ni attendre que le thread finisse, vous devriez plutôt appeler la méthode `detach()`, décrite ci-dessous.

### 46.8.6 Ignorer un thread

`join()` fait trois choses : il attend qu'un thread se termine, le nettoie, et retourne les valeurs qu'il peut avoir produites. Mais que faire si vous n'êtes pas intéressé par le retour du thread, et que vous n'avez cure de quand il finit ? Tout ce qui vous importe alors est que le thread soit nettoyé quand il se termine.

Dans ce cas, utilisez la méthode `detach()`. Une fois qu'un thread est détaché, il continuera à s'exécuter jusqu'à ce qu'il termine, après quoi Perl le nettoiera automatiquement.

```
use threads;

$thr = threads->new(\&sub1); # Générer le thread

$thr->detach; # A partir de maintenant, nous nous désintéressons
 # officiellement du thread

sub sub1 {
 $a = 0;
 while (1) {
 $a++;
 print "\$a vaut $a\n";
 sleep 1;
 }
}
```

Une fois qu'un thread est détaché, il ne peut plus être rejoint (avec `join()`), et toute valeur de retour qu'il pourrait avoir produite (s'il avait déjà fini et attendait un `join()`) est perdue.

## 46.9 Threads et données

Maintenant que nous avons couvert les bases des threads, il est temps de passer au sujet suivant : les données. L'utilisation des threads introduit quelques complications à l'accès aux données dont les programmes sans threads n'ont jamais à se préoccuper.



### 46.9.1 Données partagées et non partagées

La plus grande différence entre les itreads de Perl et les vieux threads 5.005, ou d'ailleurs n'importe quel autre système de threads, est que par défaut, aucune donnée n'est partagée. Quand un nouveau thread Perl est créé, toutes les données associées au thread courant sont copiées vers le nouveau thread, et sont dorénavant données privées du nouveau thread ! C'est similaire dans l'esprit à ce qui arrive quand un processus UNIX fait un fork(), sauf que dans notre cas les données sont copiées vers une partie différente de la mémoire à l'intérieur du même processus, sans qu'un vrai fork() ait lieu.

Pour faire bon usage des threads, cependant, on veut généralement partager des données entre eux. On peut le faire grâce au module `threads::shared` et à l'attribut `: shared` (*partagé*).

```
use threads;
use threads::shared;

my $toto : shared = 1;
my $tata = 1;
threads->new(sub { $toto++; $tata++ }->join;

print "$toto\n"; # affiche 2 car $toto est partagé
print "$tata\n"; # affiche 1 car $tata n'est pas partagé
```

Dans le cas d'un tableau partagé, tous les éléments du tableau sont partagés, et pour une table de hachage partagée, toutes les clés et les valeurs sont partagées. Cela place des restrictions sur ce qui peut être affecté à des éléments de tableaux et de tables de hachage partagés : seules des valeurs simples ou des références à des variables partagées sont autorisées - de façon à ce qu'une variable privée ne puisse accidentellement devenir partagée. Une affectation incorrecte entraîne la mort du thread (*die*). Par exemple :

```
use threads;
use threads::shared;

my $var = 1;
my $svar : shared = 2;
my %hash : shared;

... créer quelques threads ...

$hash{a} = 1; # pour tous les threads, exists($hash{a}) et $hash{a} == 1
$hash{a} = $var; # ok - copie par valeur : même effet que précédemment
$hash{a} = $svar; # ok - copie par valeur : même effet que précédemment
$hash{a} = \$svar; # ok - référence à une variable partagée
$hash{a} = \$var; # entraîne la terminaison (I<die>)
delete $hash{a}; # ok - pour tous les threads, !exists($hash{a})
```

Remarquez qu'une variable partagée garantit que si deux threads ou plus essaient de la modifier au même moment, son état interne ne sera pas corrompu. Cependant, il n'y a pas de garantie au-delà de celle-ci, comme cela est expliqué dans la prochaine section.

### 46.9.2 Pièges des threads : *race conditions*

Note de traduction : nous utilisons le terme anglais *race condition* pour désigner un cas où le comportement d'un programme dépend de l'ordre d'événements particuliers, alors que cet ordre ne peut pas être garanti.

Les threads apportent des outils très utiles, mais amènent aussi leur lot de pièges. L'un de ces pièges est la *race condition*.

```
use threads;
use threads::shared;

my $a : shared = 1;
$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub2);
```

```

$thr1->join;
$thr2->join;
print "$a\n";

sub sub1 { my $toto = $a; $a = $toto + 1; }
sub sub2 { my $tata = $a; $a = $tata + 1; }

```

Quelle sera la valeur de \$a, d'après vous ? Par chance, la réponse est "ça dépend". sub1() et sub2() accèdent toutes les deux à la variable globale \$a, une fois en lecture et une fois en écriture. Selon différents facteurs comme l'algorithme d'ordonnement de votre implémentation des threads ou la phase de la lune, \$a peut être 2 ou 3.

Les race conditions ont pour cause un accès non synchronisé à des données partagées. Sans synchronisation explicite, il n'y a aucune façon d'être sûr que rien n'arrive aux données partagées entre le moment où vous les lisez et celui où vous les modifiez. Même un bout de code aussi simple que l'exemple suivant présente un problème :

```

use threads;
my $a : shared = 2;
my $b : shared;
my $c : shared;
my $thr1 = threads->create(sub { $b = $a; $a = $b + 1; });
my $thr2 = threads->create(sub { $c = $a; $a = $c + 1; });
$thr1->join;
$thr2->join;

```

Deux threads accèdent à \$a. Chacun peut potentiellement être interrompu n'importe quand, et ils peuvent être exécutés dans n'importe quel ordre. A la fin, \$a peut valoir 3 ou 4, et \$b et \$c peuvent chacun valoir 2 ou 3.

Même pour \$a += 5 et \$a++, l'atomicité n'est pas garantie.

A chaque fois que votre programme accède à des données ou des ressources auxquelles peuvent aussi accéder d'autres threads, vous devez prendre des mesures pour assurer la coordination des accès, ou courir le risque de vous retrouver avec des données inconsistantes ou des race conditions. Remarquez que Perl protège son état interne contre vos race conditions, mais il ne vous protège pas de vous-même.

## 46.10 Synchronisation et contrôle

Perl fournit plusieurs mécanismes pour coordonner les interactions entre les threads et leurs données, et pour éviter les race conditions. Certains ressemblent aux techniques communes des bibliothèques de gestion des threads, comme pthreads ; d'autres sont spécifiques à Perl. Le plus souvent, les techniques standards (comme les attentes de condition) sont maladroites et difficiles à mettre en place correctement. Quand c'est possible, il est généralement plus facile d'utiliser les procédés perliens comme les files d'attente, qui se chargent d'une partie du travail pénible.

### 46.10.1 Contrôler l'accès : lock()

La fonction lock() prend une variable partagée et la verrouille. Aucun autre thread ne peut la verrouiller jusqu'à ce que la variable ait été déverrouillée par le thread qui a posé le verrou. Le déverrouillage arrive automatiquement quand le thread qui tient le verrou sort du premier bloc extérieur qui contient l'appel à la fonction lock(). L'utilisation de lock() est simple ; l'exemple suivant fait faire des calculs en parallèle à plusieurs threads, en mettant à jour un total courant de temps en temps :

```

use threads;
use threads::shared;

my $total : shared = 0;

sub calc {
 for (;;) {
 my $resultat;
 # (... faire des calculs et affecter à $resultat ...)
 {
 lock($total); # bloque jusqu'à obtenir le verrou

```

```

 $total += $resultat;
 } # le verrou est automatiquement relâché à la sortie du bloc
 last if $resultat == 0;
}

my $thr1 = threads->new(\&calc);
my $thr2 = threads->new(\&calc);
my $thr3 = threads->new(\&calc);
$thr1->join;
$thr2->join;
$thr3->join;
print "total=$total\n";

```

lock() bloque le thread jusqu'à ce que la variable à verrouiller soit disponible. Quand lock() retourne, votre thread peut être sûr qu'aucun autre ne peut verrouiller la variable jusqu'à la fin du premier bloc extérieur contenant l'appel à lock().

Il est important de noter que les verrous n'empêchent pas l'accès à la variable en question, mais seulement les tentatives de verrouillage. Cela s'inscrit dans la longue amitié qu'à Perl avec la programmation bien élevée, ainsi que le verrouillage consultatif (*advisory locking*) de fichiers que flock() permet.

Vous pouvez verrouiller des tableaux et des tables de hachage, aussi bien que des scalaires. Cependant, le verrouillage d'un tableau ne bloque pas d'éventuels verrouillages ultérieurs sur ses éléments, mais juste ceux sur le tableau lui-même.

Les verrous sont récursifs, ce qui veut dire qu'un thread peut verrouiller une variable plus d'une fois. Le verrou durera jusqu'à ce que le lock() le plus extérieur sur la variable arrive au bout de sa portée. Par exemple :

```

my $x : shared;
fais();

sub fais {
 {
 {
 lock($x); # attendre le verrouillage
 lock($x); # NOOP - nous avons déjà le verrou
 {
 lock($x); # NOOP
 {
 lock($x); # NOOP
 verrouille_le_encore();
 }
 }
 } # *** déverrouillage implicite ici ***
 }
}

sub verrouille_le_encore {
 lock($x); # NOOP
} # rien n'arrive ici

```

Notez qu'il n'y a pas de fonction unlock() - la seule façon de déverrouiller une variable est de laisser le verrou sortir de sa portée.

Un verrou peut être utilisé pour préserver les données contenues dans la variable verrouillée, mais il peut aussi servir à garder autre chose, comme une section de code. Dans ce cas, la variable verrouillée ne contient pas de données utiles, et n'existe que pour être verrouillée. La variable se comporte alors comme une mutex ou un sémaphore simple des bibliothèques traditionnelles de gestion des threads.

### 46.10.2 Un piège des threads : interblocages (*deadlocks*)

Les verrous sont un outil pratique pour synchroniser les accès aux données, et leur bon usage est la clé de la sûreté des données partagées. Malheureusement, ils ne sont pas sans danger, en particulier quand plusieurs verrous sont en jeu. Contemplez le code suivant :

```

use threads;

my $a : shared = 4;
my $b : shared = "toto";
my $thr1 = threads->new(sub {
 lock($a);
 threads->yield;
 sleep 20;
 lock($b);
});
my $thr2 = threads->new(sub {
 lock($b);
 threads->yield;
 sleep 20;
 lock($a);
});

```

Ce programme va probablement se bloquer jusqu'à ce que vous le tuiez. La seule façon qu'il ne se bloque pas est qu'un des deux threads acquière les deux verrous en premier. Une version garantie de bloquer serait plus compliquée, mais le principe serait le même.

Le premier thread va acquérir un verrou sur \$a, puis, après une pause durant laquelle le second thread a probablement eu le temps de travailler, essayer de verrouiller \$b. Pendant ce temps, le second thread acquiert un verrou sur \$b, puis plus tard essaie de verrouiller \$a. La deuxième tentative de verrouillage pour les deux threads va bloquer, chacun attendant que l'autre relâche son verrou.

Cette situation est appelée un interblocage, et elle arrive dès que deux threads ou plus essaient d'obtenir un verrou sur des ressources que les autres ont verrouillées. Chaque thread va bloquer, en attendant que l'autre relâche un verrou sur une ressource. Cependant, cela n'arrive jamais puisque le thread qui détient la ressource est lui-même en train d'attendre qu'un autre verrou soit relâché.

Il y a plusieurs façon de régler ce genre de problèmes. La meilleure est de s'assurer que tous les threads acquièrent les verrous dans le même ordre. Si, par exemple, vous verrouillez les variables \$a, \$b et \$c, verrouillez toujours \$a avant \$b, et \$b avant \$c. Il est aussi préférable de ne garder les verrous que pendant une courte période de temps pour minimiser les risques d'interblocage.

Les autres primitives de synchronisation décrites plus bas peuvent souffrir de problèmes similaires.

### 46.10.3 Files d'attente (*queues*) : transmettre des données

Une file d'attente est un objet spécial qui vous permet d'enfourner des données d'un côté et de les récupérer de l'autre, sans avoir à vous inquiéter des problèmes de synchronisation. C'est un animal assez simple, qui ressemble à ceci :

```

use threads;
use Thread::Queue;

my $File = Thread::Queue->new;
$thr = threads->new(sub {
 while ($Element = $File->dequeue) {
 print "Retiré $Element de la file\n";
 }
});

$File->enqueue(12);
$File->enqueue("A", "B", "C");
$File->enqueue(\$thr);
sleep 10;
$File->enqueue(undef);
$thr->join;

```

Vous pouvez créer une file d'attente avec `new Thread::Queue`. Ensuite, vous pouvez y empiler des listes de scalaires avec `enqueue()`, et en retirer des scalaires de l'autre côté avec `dequeue()`. Une file n'a pas de taille fixe, et peut grandir à volonté pour loger tout ce qui y est poussé.

Si une file est vide, `dequeue()` bloque jusqu'à ce qu'un autre thread y pousse quelque chose. Cela fait des files l'élément idéal pour les boucles d'événements et les autres sortes de communication entre threads.

#### 46.10.4 Sémaphores : synchroniser les accès aux données

Les sémaphores sont un genre de mécanisme de verrouillage générique. Dans leur forme la plus basique, ils se comportent tout à fait comme des scalaires munis d'un verrou, sauf qu'ils ne peuvent pas contenir de données, et doivent être explicitement déverrouillés. Dans leur forme avancée, ils agissent comme un genre de compteur, et peuvent autoriser plusieurs threads à détenir le "verrou" à un instant donné.

#### 46.10.5 Sémaphores de base

Les sémaphores ont deux méthodes, `down()` et `up()` : `down()` décrémente le compteur de ressource, `up()` l'incrmente. Les appels à `down()` bloqueront si le compteur courant du sémaphore devait descendre sous zéro pour effectuer le `down()`. Ce programme en donne une rapide démonstration :

```
use threads qw(yield);
use Thread::Semaphore;

my $semaphore = new Thread::Semaphore;
my $VariableGlobale : shared = 0;

$thr1 = new threads \&sub_exemple, 1;
$thr2 = new threads \&sub_exemple, 2;
$thr3 = new threads \&sub_exemple, 3;

sub sub_exemple {
 my $NoSub = shift @_;
 my $NbTentatives = 10;
 my $CopieLocale;
 sleep 1;
 while ($NbTentatives-->0) {
 $semaphore->down;
 $CopieLocale = $VariableGlobale;
 print "$NbTentatives essais restants pour la sub"
 . " $NoSub (\$VariableGlobale est $VariableGlobale)\n";
 yield;
 sleep 2;
 $CopieLocale++;
 $VariableGlobale = $CopieLocale;
 $semaphore->up;
 }
}

$thr1->join;
$thr2->join;
$thr3->join;
```

Les trois appels de la fonction opèrent tous en même temps. Le sémaphore, cependant, s'assure qu'il n'y a qu'un thread à la fois qui accède à la variable globale.

#### 46.10.6 Sémaphores avancés

Par défaut, les sémaphores se comportent comme des verrous, et laissent un seul thread à la fois décrémente leur compteur avec `down()`. On peut cependant en faire d'autres usages.

Chaque sémaphore a un compteur attaché. Par défaut, les sémaphores sont créés avec un compteur mis à un, `down()` décrémente ce compteur d'une unité, et `up()` l'incrmente d'une unité. Nous pouvons néanmoins utiliser d'autres valeurs de la façon suivante :

```
use threads;
use Thread::Semaphore;
my $semaphore = Thread::Semaphore->new(5);
 # Crée un sémaphore avec le compteur initialisé
 # à cinq
```

```

$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub1);

sub sub1 {
 $semaphore->down(5); # Décrémente le compteur de cinq unités
 # Faire des choses ici
 $semaphore->up(5); # Incrémente le compteur de cinq unités
}

$thr1->detach;
$thr2->detach;

```

Si `down()` essaie de décrémenter le compteur au-dessous de zéro, il bloque jusqu'à ce que le compteur devienne assez grand. Notez que bien qu'un sémaphore puisse être créé avec un compteur initial de zéro, tout `up()` ou `down()` change toujours le compteur d'au moins une unité ; par conséquent, `$semaphore->down(0)` est identique à `$semaphore->down(1)`.

La question, bien sûr, est pourquoi voudrait-on faire ça ? Pourquoi créer un sémaphore avec un compteur initial qui n'est pas égal à un, ou pourquoi décrémenter/incrémenter par plus d'une unité ? La réponse tient dans la disponibilité des ressources. De nombreuses ressources auxquelles on veut gérer l'accès peuvent être utilisées par plusieurs threads en même temps en toute sécurité.

Par exemple, considérons un programme centré autour d'une interface graphique. Il utilise un sémaphore pour synchroniser l'accès à l'affichage, de manière à ce qu'un seul thread à la fois puisse dessiner. C'est pratique, mais bien sûr vous ne voulez pas qu'un thread commence à dessiner avant que tout ne soit mis en place. Dans ce cas, vous pouvez créer un sémaphore avec un compteur initial de zéro, et l'incrémenter quand tout est prêt pour le dessin.

Les sémaphores avec un compteur plus grand que un sont utiles pour établir des quotas. Supposons, par exemple, que vous avez plusieurs threads qui peuvent faire des E/S en parallèle. Vous ne voulez pas que tous lisent et écrivent en même temps, car cela pourrait encombrer vos canaux d'E/S, ou épuiser le quota de descripteurs de fichiers de votre processus. Vous pouvez utiliser un sémaphore initialisé au nombre de requêtes d'E/S (ou d'ouvertures de fichiers) concurrentes que vous voulez autoriser à un instant donné, et laisser tranquillement vos threads se bloquer et se débloquer les uns les autres.

Les incréments et décréments plus grands que un sont utiles quand un thread doit libérer ou accéder à plusieurs ressources à la fois.

### 46.10.7 `cond_wait()` et `cond_signal()`

Ces deux fonctions peuvent être utilisées en conjonction avec les verrous pour notifier des threads qui coopèrent qu'une ressource est devenue disponible. Elles sont très semblables aux fonctions qu'on peut trouver dans `pthread`. Mais dans la plupart des cas, les files d'attente sont plus simples à utiliser et plus intuitives. Voyez `threads::shared` pour plus de détails.

## 46.11 Fonctions utiles générales

Nous avons couvert les parties principales du système de gestion des threads de Perl, et avec ces outils vous devriez être bien équipé pour écrire du code et des modules utilisant les threads. Mais il reste quelques petits éléments que nous n'avons pas abordés.

### 46.11.1 Dans quel thread suis-je ?

La méthode de classe `threads->self` fournit à votre programme une façon d'obtenir un objet qui représente le thread dans lequel il s'exécute. Vous pouvez utiliser cet objet de la même façon que ceux renvoyés par les fonctions de création de threads.

### 46.11.2 Identificateur de thread

`tid()` est une méthode d'objet sur les threads qui renvoie l'identificateur du thread que l'objet représente. Les identificateurs sont des entiers, et celui du thread principal d'un programme est 0. Actuellement, Perl affecte un `tid` unique à chaque thread créé dans votre programme ; le premier thread crée reçoit un `tid` de 1, puis chaque nouveau thread reçoit un `tid` incrémenté de 1.

### 46.11.3 Est-ce que ces threads sont les mêmes ?

La méthode `equal()` prend deux objets `thread` et renvoie vrai si les objets représentent le même thread, faux sinon.

La comparaison avec `==` est surchargée pour les objets threads, et vous pouvez donc comparer ceux-ci comme vous le faites pour les objets ordinaires.

### 46.11.4 Quels threads sont en train de tourner ?

`threads->list` renvoie une liste d'objets threads, un pour chaque thread actuellement en train de tourner et non détaché. C'est utile à plusieurs fins, dont le nettoyage à la fin de votre programme :

```
Parcourir la liste de tous les threads
foreach $thr (threads->list) {
 # Ne pas rejoindre le thread principal ni nous-mêmes
 if ($thr->tid && !threads::equal($thr, threads->self)) {
 $thr->join;
 }
}
```

Si certains threads n'ont pas fini de tourner quand le thread Perl principal se termine, Perl vous avertit et meurt, puisqu'il est impossible à Perl de sortir proprement alors que d'autres threads tournent encore.

## 46.12 Un exemple complet

Déjà embrouillé ? Il est temps qu'un exemple illustre une partie de ce que nous avons exposé. Ce programme trouve des nombres premiers en utilisant des threads.

```
1 #!/usr/bin/perl -w
2 # prime-pthread, avec l'aimable autorisation de Tom Christiansen
3
4 use strict;
5
6 use threads;
7 use Thread::Queue;
8
9 my $flot = new Thread::Queue;
10 my $enfant = new threads(\&teste_nombre, $flot, 2);
11
12 for my $i (3 .. 1000) {
13 $flot->enqueue($i);
14 }
15
16 $flot->enqueue(undef);
17 $enfant->join;
18
19 sub teste_nombre {
20 my ($amont, $premier_courant) = @_;
21 my $enfant;
22 my $aval = new Thread::Queue;
23 while (my $nombre = $amont->dequeue) {
24 next unless $nombre % $premier_courant;
25 if ($enfant) {
26 $aval->enqueue($nombre);
27 } else {
28 print "Ai trouvé le premier $nombre\n";
29 $enfant = new threads(\&teste_nombre, $aval, $nombre);
30 }
31 }
32 $aval->enqueue(undef) if $enfant;
33 $enfant->join if $enfant;
34 }
```

Ce programme utilise le modèle de travail à la chaîne pour générer des nombres premiers. Chaque thread dans la chaîne a une file d'entrée qui fournit des nombres à tester, un nombre premier dont il est responsable, et une file de sortie dans laquelle il enfile les nombres qui ont raté son test (de non primauté). Si le thread tombe sur un nombre qui a raté le test et qu'il n'y a pas de thread enfant, c'est qu'il doit avoir trouvé un nouveau nombre premier. Dans ce cas, un nouveau thread enfant est créé pour ce premier, et accolé au bout de la chaîne.

Cela sonne probablement un peu plus difficile que cela ne l'est vraiment ; passons donc en revue ce programme morceau par morceau, et examinons ce qu'il fait. (Pour ceux qui essaient de se rappeler ce qu'est au juste un nombre premier, c'est un nombre qui est seulement divisible par 1 et lui-même.)

Le gros du travail est fait par la fonction `teste_nombre`, qui prend une référence à sa file d'entrée et un nombre premier dont elle est responsable. Après avoir extrait la file et le nombre premier pour lequel elle fait les tests (ligne 20), nous créons une nouvelle file (ligne 22) et nous réservons un scalaire pour le thread que nous allons sans doute créer plus loin (ligne 21).

La boucle `while` de la ligne 23 à la ligne 31 retire un scalaire de la file d'entrée et le teste avec le nombre premier dont le thread est responsable. La ligne 24 détermine s'il y a un reste quand nous calculons le nombre à tester modulo notre premier. S'il y en a un, le nombre n'est pas divisible par notre premier, et nous devons donc soit le passer au thread suivant si nous en avons créé un (ligne 26) soit créer un nouveau thread si ce n'est pas encore fait.

La création du nouveau thread se trouve ligne 29. Nous passons une référence à la file que nous avons créée et le nombre premier que nous avons trouvé.

Enfin, une fois que la boucle se termine (parce que nous avons trouvé un 0 ou un undef dans la file, qui sert comme un ordre de terminer), nous notifions notre enfant et attendons qu'il sorte si nous en avons créé un (lignes 32 et 37).

Pendant ce temps, dans le thread principal, nous créons une file (ligne 9) et le thread enfant initial (ligne 10), et nous lui passons le premier nombre premier : 2. Ensuite, nous enfilons tous les nombres de 3 à 1000 pour qu'il soient testés (lignes 12-14), puis une notification de terminer (ligne 16), et nous attendons que le premier thread enfant finisse (ligne 17). Nous savons que tout va bien quand nous retournons du `join()`, puisqu'un enfant ne mourra pas avant que ses propres enfants ne soient morts.

Voilà tout. C'est très simple ; comme beaucoup de programmes Perl, l'explication est bien plus longue que le programme.

## 46.13 Considérations de performance

La chose la plus importante à garder à l'esprit quand on compare les `ithreads` à d'autres modèles de gestion des threads est le fait que pour chaque thread créé, une copie complète de toutes les variables et données du thread parent doit être faite. La création d'un thread peut donc être coûteuse en termes de mémoire autant que de temps de calcul. La meilleure façon de réduire ces coûts est d'avoir un faible nombre de threads qui vivent longtemps, tous créés assez tôt - avant que le thread de base n'ait accumulé trop de données en mémoire. Bien sûr, après qu'un thread a été créé, ses performances et son occupation mémoire devraient être peu différentes que celles de code ordinaire.

Notez aussi que dans l'implémentation actuelle, les variables partagées prennent un peu plus de mémoire et sont légèrement plus lentes que les variables ordinaires.

## 46.14 Changements au niveau du processus

Bien que les threads aient des chemins d'exécution séparés et que les données Perl soient privées à chaque thread à moins qu'elles ne soient explicitement partagées, les threads peuvent modifier l'état du processus complet, affectant ainsi les autres threads.

L'exemple évident de cela est le changement du répertoire de travail courant avec `chdir()`. Si un thread appelle `chdir()`, le répertoire de travail de tous les threads change.

Un exemple encore plus spectaculaire d'un changement qui intervient au niveau du processus est `chroot()` : le répertoire racine de tous les threads change alors, et aucun thread ne peut revenir en arrière (par opposition à `chdir()`).

`umask()` et les changements d'`uid` et de `gid` sont d'autres exemples de changement qui ont lieu au niveau du processus.

Si jamais l'envie vous prenait de mélanger `fork()` et les threads, allongez-vous et attendez que ça passe. Mais si vous voulez vraiment savoir, la sémantique est que `fork()` duplique tous les threads (sous UNIX du moins - les autres plateformes feront quelque chose de différent).

De même, vous ne devriez pas essayer de mélanger les signaux et les threads. Les implémentations dépendent de la plateforme, et même la sémantique POSIX pourrait vous surprendre (sans compter que Perl ne vous fournit pas l'API POSIX complète).



## 46.15 Réentrance des bibliothèques système

La réentrance des différents appels système est hors du contrôle de Perl. Parmi les appels connus pour ne pas être réentrants, on trouve : `localtime()`, `gmtime()`, `get{gr,host,net,proto,serv,pw}*()`, `readdir()`, `rand()`, et `srand()`, et en général, tous les appels qui dépendent d'un état externe global.

Si le système sur lequel perl est compilé a des versions réentrantes de ces fonctions, elles seront utilisées. A part ça, Perl est à la merci de la réentrance ou de la non-réentrance de ces fonctions. Consultez la documentation de votre bibliothèque C pour plus de détails.

Sur certaines plateformes, l'interface réentrante peut échouer si le tampon de résultat est trop petit (par exemple `getgrent()` peut renvoyer d'assez volumineuses listes de membres). Perl réessaie alors un certain nombre de fois en agrandissant le tampon, mais jusqu'à 64ko seulement (pour des raisons de sécurité).

## 46.16 Conclusion

Un tutoriel complet sur les threads pourrait remplir un livre entier (en fait, de nombreux livres ont pour seul objet ce type de tutoriel), mais avec ce que nous avons couvert dans cette introduction, vous devriez être en bonne voie pour devenir un expert de la programmation avec les threads en Perl.

## 46.17 Bibliographie

Voici une courte bibliographie, aimablement fournie par Jürgen Christoffel.

### 46.17.1 Textes introductifs

Birrell, Andrew D. *An Introduction to Programming with Threads*. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 en ligne sur <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-035.html> (hautement recommandé)

Robbins, Kay. A., et Steven Robbins. *Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice-Hall, 1996.

Lewis, Bill, et Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1997, ISBN 0-13-443698-9 (une introduction aux threads très bien écrite).

Nelson, Greg (sous la direction de). *Systems Programming with Modula-3*. Prentice Hall, 1991, ISBN 0-13-590464-1.

Nichols, Bradford, Dick Buttlar, et Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996, ISBN 156592-115-1 (couvre les threads POSIX).

### 46.17.2 Références liées aux systèmes d'exploitation

Boykin, Joseph, David Kirschen, Alan Langerman, et Susan LoVerso. *Programming under Mach*. Addison-Wesley, 1994, ISBN 0-201-52739-1.

Tanenbaum, Andrew S. *Distributed Operating Systems*. Prentice Hall, 1995, ISBN 0-13-219908-4 (très bon manuel).

Silberschatz, Abraham, et Peter B. Galvin. *Operating System Concepts*, 4th ed. Addison-Wesley, 1995, ISBN 0-201-59292-4

### 46.17.3 Autres références

Arnold, Ken et James Gosling. *The Java Programming Language*, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.  
FAQ de `comp.programming.threads`, <http://www.serpentine.com/~bos/threads-faq/>

Le Sergent, T. et B. Berthomieu. "Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures" in *Memory Management : Proc. of the International Workshop IWMM 92*, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540-55940-X (applications pratiques des threads).

Arthur Bergman, "Where Wizards Fear To Tread", June 11, 2002, <http://www.perl.com/pub/a/2002/06/11/threads.html>

## 46.18 Crédits

Merci (sans ordre particulier) à Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, et Alan Burlison, pour leur aide dans la vérification et la mise au point de cet article. Un grand merci à Tom Christiansen pour sa réécriture du générateur de nombres premiers.

## 46.19 AUTEUR

Dan Sugalski (dan@sidhe.org).

Légèrement modifié par Arthur Bergman pour ajuster l'article au nouveau modèle de gestion des threads.

Légèrement retravaillé par Jörg Walter <jwalt@cpan.org> pour être plus concis sur le sujet de la réentrance du code Perl.

## 46.20 Copyrights

The original version of this article originally appeared in The Perl Journal #10, and is copyright 1998 The Perl Journal ( <http://www.tpj.com> ). It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

## 46.21 Copyright

La version originale de cet article est parue dans le numéro 10 de The Perl Journal, et est copyright 1998 The Perl Journal ( <http://www.tpj.com> ). Il est utilisé ici avec l'aimable autorisation de Jon Orwant et de The Perl Journal. Ce document peut être redistribué sous les mêmes termes que Perl lui-même.

Pour plus d'informations, voyez *threads* et *threads::shared*.

## 46.22 TRADUCTION

### 46.22.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 46.22.2 Traducteur

Ronan Le Hy <rlehy@free.fr>.

### 46.22.3 Relecture

Personne pour l'instant.

# Chapitre 47

## perlport

Écrire du code Perl portable

### 47.1 DESCRIPTION

Perl fonctionne sur une grande variété de système d'exploitation. Alors que la plupart d'entre eux ont beaucoup en commun, il ont aussi des fonctionnalités qui leur sont très spécifiques et uniques.

Ce document tente de vous aider à trouver ce qui constitue un code Perl portable, de manière que si vous avez décidé d'écrire du code portable, vous sachiez où sont les limites et ainsi ne pas les franchir.

C'est un compromis entre tirer pleinement avantage d'un type de machine particulier, et tirer avantage d'une **quantité** d'entre eux. Bien sûr, plus votre quantité est large (et ainsi plus variée), plus le nombre de dénominateurs commun chute, vous laissant un choix plus restreint de solution pour accomplir une tâche précise. Il est donc important, avant d'attaquer un problème, de considérer l'enveloppe des compris que vous êtes prêt à accepter. Précisément, s'il y est important pour vous que la tâche nécessite une portabilité totale, ou s'il est seulement important qu'elle soit réalisée. C'est le choix le plus difficile à faire. Le reste est facile, parce que Perl fournit un très grand nombre de possibilités quelque soit l'approche que vous avez de votre problème.

D'un autre sens, écrire du code portable va inévitablement réduire les choix disponibles. C'est une discipline à acquérir.

Faites attention aux points importants:

#### **Tous les programmes Perl n'ont pas besoin d'être portable**

Il n'y a aucune raison de ne pas utiliser Perl comme un langage pour enchaîner l'exécution d'outil Unix, ou pour prototyper une application Macintosh, ou pour gérer la base de registre de Windows. Si la portabilité d'un programme n'a pas de sens, inutile de s'en préoccuper.

#### **L'immense majorité de Perl est portable**

N'allez pas penser qu'il est difficile d'écrire du code Perl portable. Ce n'est pas le cas. Perl essaye de réduire de lui-même les différences entre les différentes plateformes, et de fournir tout les moyens nécessaires pour utiliser ces fonctionnalités. Dès lors presque tous les scripts Perl fonctionnent sans aucune modification. Mais il *existe* quelques spécificités dans l'écriture de code portable, et ce document est entièrement consacré à ces spécificités.

Voici la règle générale: Lorsque vous entamez une tâche qui est communément réalisées sur des plateformes différentes, pensez à écrire du code portable. De cette manière, vous ne vous limitez pas trop dans le nombre de solutions possible, et dans le même temps, vous ne limitez pas l'utilisateur à un trop petit choix de plateforme. D'un autre côté, lorsque vous devez utiliser des particularités d'une plateforme spécifique, comme c'est souvent le cas pour la programmation système (Quelque soit le type de plateforme Unix, Windows, Mac OS, VMS, etc.), écrivez du code spécifique.

Lorsqu'un script doit tourner que sur deux ou trois système d'exploitation, vous n'avez qu'à considérer les différences entre ces systèmes particuliers. L'important est de décider sur quoi le script doit fonctionner et d'être clair dans votre décision.

Le document est séparé en trois sections principales: les problèmes généraux de la portabilité (PROBLÈMES (§47.2)), les spécificités par plateforme (PLATEFORMES (§47.4)), et les différences de comportement des fonctions intégrées de perl suivant les plateformes (IMPLÉMENTATION DES FONCTIONS (§47.5)).

Ces informations ne doivent pas être considérés comme exhaustives; elles peuvent décrire des comportements transitoires de certain portage, elles sont presque toutes en évolutions permanente. Elle doivent donc être considérés comme étant en cours de modification perpétuelles. (<IMG SRC="yellow\_sign.gif" ALT="Under Construction">).

## 47.2 PROBLÈMES

### 47.2.1 Les retours chariots

Dans la plupart des systèmes d'exploitations, les lignes des fichiers sont séparées par des retours chariots. Mais la composition de ces retour chariots diffèrent d'OS en OS. Unix utilise traditionnellement `\012`, un type d'entrée/sortie de Windows utilise `\015\012`, et Mac OS utilise `\015`.

Perl utilise `\n` pour représenter le retour chariot "logique", où ce qui est logique peut dépendre de la plateforme utilisée. Avec MacPerl, `\n` correspond toujours à `\015`. Pour les Perl DOS, il correspond habituellement à `\012`, mais lors d'un accès à un fichier en mode "texte", STDIO le converti vers (ou à partir de) `\015\012`.

A cause de la conversion "texte", Perl sous DOS ont des limitations à l'utilisation de `seek` et `tell` lorsqu'un fichier est utilisé en mode "texte". Spécifiquement, si vous n'utilisez que des emplacements obtenus avec `tell` (et rien d'autre) pour `seek`, vous pouvez l'utiliser sans problèmes en mode "texte". En général utiliser `seek` ou `tell` ou quelques opérations portant sur un nombre d'octet plutôt qu'un nombre de caractères, sans faire attention à la taille du `\n`, peut ne pas être portable. Si vous utilisez le mode `binmode`, ces limitations n'existent pas.

Une erreur de conception commune dans la programmation des sockets est de penser que `\n` eq `\012` partout. Lors de l'utilisation de protocoles tel que les protocoles Internet `\012` et `\015` doivent être spécifié, et les valeurs de `\n` et `\r` ne sont fiables.

```
print SOCKET "Hi there, client!\r\n"; # MAUVAIS
print SOCKET "Hi there, client!\015\012"; # BON
```

[NOTE: ceci ne concerne pas nécessairement les communications qui sont filtrées par un autre programme ou module avant d'être envoyées sur la socket; le serveur web EBCDIC le plus populaire, par exemple, accepte `\r\n`, et les convertis ainsi que tous les caractères d'un flux texte, de l'EBCDIC à l'ASCII.]

Quoi qu'il en soit, utiliser `\015\012` (ou `\cM\cJ`, ou `\x0D\x0A`) peu vite devenir fatigant et laid, et peut même induire en erreur ceux qui maintiennent le code. Le module `Socket` fournit la Bonne Chose à faire pour ceux qui le veulent.

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF" # BON
```

En lisant à *partir* d'une socket, rappelez-vous que le séparateur par défaut (`$/`) est `\n`, mais un code comme celui-ci devrait reconnaître `$/` comme `\012` ou `\015\012`:

```
while (<SOCKET>) {
 # ...
}
```

Meilleur:

```
use Socket qw(:DEFAULT :crlf);
local($/) = LF; # pas nécessaire si $/ est déjà \012

while (<SOCKET>) {
 s/$CR?$LF/\n/; # il n'est pas sur que la socket utilise LF ou CRLF, OK
 # s/\015?\012/\n/; # La même chose
}
```

Cet exemple est le meilleur que le précédent même sous Unix, car maintenant les `\015` et `(\cM` sont supprimés (et c'est tant mieux).

### 47.2.2 Stockage et taille des nombres

Différent CPU stocke les entiers et les flottants dans des ordres différents (en anglais *endianness*) et des tailles différentes (32-bit et 64-bit sont les plus courants). Ceci affecte vos programmes s'ils essaient de transférer des nombres en format binaire d'une architecture CPU vers une autre via certains canaux: que ce soit en 'live' via une connexion réseau ou en les stockant dans un fichier.

Un conflit d'ordre de stockage met le désordre dans les nombres: Si un hôte little-endian (Intel, Alpha) stocke 0x12345678 (305419896 en décimal), un hôte big-endian (Motorola, MIPS, Sparc, PA) le lira comme étant 0x78563412 (2018915346 en décimal). Pour éviter ce problème lors de communication réseau (socket) utilisez les formats "n" et "N" des fonctions `pack()` et `unpack()`, l'ordre "réseau", car il est garanti d'être portable.

Des tailles différentes provoquera des troncatures même entre des plateformes de même ordre: la plateforme de taille plus courte perdra la partie haute du nombre. Il n'y a pas de bonne solution pour ce problème sauf éviter de transférer ou stocker des nombres en format binaire.

Vous pouvez circonvenir ces deux problèmes de deux façons: Soit en transférant et stockant les nombres en format texte, et non pas en binaire, ou en utilisant des modules tel que `Data::Dumper` (inclus dans la distribution standard depuis Perl 5.005) et `Storable`.

### 47.2.3 Fichiers

La plupart des plateformes de nos jours stockent les fichiers d'une manière hiérarchique. Alors, il est raisonnable de penser que n'importe quelle plateforme supporte une notion de "chemin" pour identifier de manière unique un fichier sur le système. La seule chose qui change est l'écriture de ce chemin.

Quoique similaire, les spécifications de chemin diffèrent entre Unix, Windows, Mac OS, OS/2, VMS, RISC OS et probablement les autres. Unix, par exemple, est l'un des rares OS à avoir une idée d'un répertoire racine unique.

VMS, Windows, et OS/2 peuvent fonctionner comme Unix en utilisant / comme séparateur de chemin, ou de leur propre façon (tel qu'avoir plusieurs répertoires racines et des fichiers périphériques non-rattachés à une racine tel que NIL: et LPT:).

<Mac OS> utilise : comme séparateur à la place de /.

Le perl RISC OS peut émuler les noms de fichiers Unix avec / comme séparateur, ou utiliser le séparateur natif . pour les chemins et : pour le système de complétion de signal et les noms de disque.

Comme pour le problème des retours chariot, il existe des modules qui peuvent aider. Le module `File::Spec` fournit les méthodes permettant de faire la Bonne Chose quelque soit la plateforme sur laquelle le programme fonctionne.

```
use File::Spec;
chdir(File::Spec->updir()); # Remonter au répertoire parent
$file = File::Spec->catfile(
 File::Spec->curdir(), 'temp', 'file.txt'
);
sur Unix et Win32, './temp/file.txt'
sur Mac OS, ':temp:file.txt'
```

`File::Spec` est disponible dans la distribution standard, depuis la version 5.004\_05.

En général, un programme en production ne doit pas contenir de chemin codé en dur; les demander à l'utilisateur ou les obtenir d'un fichier de configuration est meilleur, en gardant à l'esprit que la syntaxe des chemins varie d'une machine à l'autre.

Ceci est particulièrement visible dans des scripts tels que Makefiles et suites de tests qui supposent souvent que / est le séparateur pour les sous-répertoire.

Toujours dans la distribution standard `File::Basename` peut aussi être utile en décomposant un chemin en morceau (nom du fichier, chemin complet et suffixe du fichier).

Même sur une plateforme unique (si l'on peut considérer Unix comme étant une plateforme unique), rappelez-vous de ne pas compter sur l'existence ou le contenu de fichier spécifique au système, tels que `/etc/passwd`, `/etc/sendmail.conf`, ou `/etc/resolv.conf`. Par exemple `/etc/passwd` peut exister mais ne pas contenir les mots de passe cryptés parce que le système utilise une forme de sécurité plus performante – ou il peut ne pas contenir tous les comptes si le système utilise NIS. Si le code doit reposer sur de tels fichiers, incluez une description des fichiers et de leur format dans la documentation, et faites en sorte que l'utilisateur puisse facilement modifier l'emplacement par défaut du fichier.

N'ayez pas deux fichiers portant le même nom dans des casses différentes, comme *test.pl* et *<Test.pl>*, car de nombreuses plateformes sont insensibles à la casse pour les noms de fichier. De plus, essayez de ne pas avoir de caractères non alphanumériques (excepté par *.*) dans les noms de fichier, et conservez la convention 8.3, pour une portabilité maximale.

De même, si vous utilisez *AutoSplit*, essayez de conserver ces conventions pour la fonction *split*; ou, au minimum, faites en sorte que le fichier résultat est un nom unique (indépendamment de la casse) pour les 8 premiers caractères.

Ne présumez pas que *<* ne sera jamais le premier caractère d'un nom de fichier. Utilisez toujours *>* explicitement pour ouvrir un fichier en lecture.

```
open(FILE, "<$existing_file") or die $!;
```

#### 47.2.4 Interaction avec le système

Toutes les plateformes ne fournissent pas forcément la notion de ligne de commande. Ce sont généralement des plateformes qui reposent sur une interface graphique (GUI) pour les interactions avec l'utilisateur. Un programme nécessitant une ligne de commande n'est donc pas forcément portable partout. Mais c'est probablement à l'utilisateur de s'arranger avec ça.

Certaines plateformes ne peuvent détruire ou renommer des fichiers ouverts par le système. Pensez donc à toujours fermer (*close*) les fichiers lorsque vous en avez terminé avec eux. N'effacez pas (*unlink*) ou ne renommez pas (*rename*) un fichier ouvert. N'utilisez pas *tie* ou *open* sur un fichier déjà ouvert ou connecté; utilisez d'abord *untie* ou *close*.

N'ouvrez jamais un même fichier plus d'une fois en même temps en écriture, car certains systèmes d'exploitation posent des verrous sur de tels fichiers.

Ne comptez pas sur la présence d'une variable d'environnement spécifique dans *%ENV*. Ne vous attendez pas à ce que les entrées de *%ENV* soit sensibles à la casse, où même ne respecte correctement celle-ci.

Ne comptez pas sur les signaux.

Ne comptez pas sur la complétion des noms de fichier. Utilisez *opendir*, *readdir*, et *closedir* à la place.

Ne comptez pas sur une copie privée des variables d'environnement par programme, ni même sur un répertoire courant différent par programmes.

#### 47.2.5 Communication inter-processus (IPC)

En général, n'accédez jamais directement au système dans un code qui se doit d'être portable. Ça signifie pas de *system*, *exec*, *fork*, *pipe*, *"*, *qx//*, *open* avec un *|*, ni aucune des autres choses qui font qu'être un hacker perl sous Unix vaille le coup.

Les commandes qui lancent des processus externes sont en général supportées par la plupart des plateformes (même si beaucoup d'entre elles ne supportent pas d'exécution simultanée (*fork* en anglais)), mais les problèmes avec elles c'est ce que l'on exécute grâce à elles. Les outils ont très souvent des noms différents suivant les plateformes, ou ne sont pas au même endroit, et en général affichent leur résultat différemment suivant la plateforme. Vous pouvez donc rarement compter sur eux pour produire un résultat consistant.

Un exemple courant de code Perl est d'ouvrir un tube vers *sendmail*:

```
open(MAIL, '|/usr/lib/sendmail -t') or die $!;
```

Ceci est correct lorsque l'on sait que *sendmail* sera disponible. Mais ceci n'est pas valable sur beaucoup de systèmes non-Unix, et même sous certains Unix qui n'auraient pas *sendmail* installé. Si vous recherchez une solution portable, jetez un œil aux modules *Mail::Send* et *Mail::Mailer* de la distribution *MailTools*. *Mail::Mailer* fournit plusieurs méthodes pour poster du courrier, incluant *mail*, *sendmail* et une connexion directe SMTP (via *Net::SMTP*) si votre système ne comporte pas de transporteur de courrier (MTA en anglais).

La règle à retenir pour du code portable est : faites-le entièrement en Perl portable, ou utilisez un module (qui peut être implémenté en utilisant du code plateforme-dépendant, mais qui présente une interface commune).

Les IPC System V (*msg\**()), *sem\**()), *shm\**()) ne sont pas toujours disponibles, même pas sur toutes les plateformes Unix.

#### 47.2.6 Sous-programme externe (XS)

Le code XS peut, en général, être écrit de manière à fonctionner sur n'importe quelle plateforme; mais alors il ne doit pas utiliser des bibliothèques, des fichiers d'inclusion (*header*), etc., dépendants, ou non disponibles sur toutes les plateformes, le code XS peut lui-même être spécifique à une plateforme, au même titre que du code Perl. Si les bibliothèques et les *headers* sont portables, il est alors raisonnable de s'assurer que le code XS est portable aussi.

Plusieurs problèmes peuvent surgir lors de l'écriture de code XS portable: la disponibilité d'un compilateur C sur le système de l'utilisateur final. Le C porte en lui ses propres problèmes de portabilité et écrire du code XS vous expose à certains d'entre eux. Écrire en perl pure est comparativement un moyen plus simple d'assurer la portabilité.

### 47.2.7 Modules Standard

En général, les modules standard fonctionnent sur toutes les plateformes. L'exception notable est `CPAN.pm` (qui utilise actuellement des programmes externes qui peuvent ne pas être disponible, les modules spécifiques à une plateforme (tels que `ExtUtils::MM_VMS`), et les modules DBM.

Il n'y a aucun module DBM qui soit disponible sur toutes les plateformes. `SDBM_File` et les autres sont en général disponible sur tous les Unix et les portages DOS, mais pas avec MacPerl, ou seul `NBDM_File` et `DB_File` sont disponibles.

La bonne nouvelle est qu'au moins un module DBM doit être disponible, et que le module `AnyDBM_File` utilisera le module DBM qu'il pourra trouver. Bien sûr, le code doit être plus strict, retombant sur un dénominateur commun (par ex., ne pas excéder 1K pour chaque enregistrement).

### 47.2.8 Date et Heure

Les notions de l'heure du jour et de la date calendaire du système sont gérés de manière très différente. Ne pensez pas que la zone horaire soit stocké dans `$ENV{TZ}`, et même si c'est le cas, ne pensez pas que vous pouvez contrôler la zone horaire grâce à cette variable.

Ne pensez pas que le temps commence le 1er Janvier 1970 à 00:00:00, car ceci est spécifique à un système d'exploitation. Le mieux est de stocker les dates dans un format non ambigu. La norme ISO 8601 défini le format de la date: AAAA-MM-JJ. Une représentation texte (comme 1 Jan 1970) peut facilement être converti en une valeur spécifique à l'OS à l'aide d'un module comme `Date::Parse`. Un tableau de valeur, tel que celui retourné par `localtime`, peut facilement être converti en une représentation spécifique à l'OS en utilisant `Time::Local`.

### 47.2.9 Jeux de caractère et encodage des caractères.

Présumez que très peu de chose sur les jeux de caractères. Ne présumez rien du tout sur les valeurs numériques (`ord()`, `chr()`) des caractères. Ne présumez pas que les caractères alphabétiques sont encodé de manière continue (au sens numérique du terme). Ne présumez rien sur l'ordre des caractères. Les minuscules peuvent être avant ou après les majuscules, elles peuvent être enlacé de telle sorte que le a et le A soit avant le b, les caractères accentués peuvent même être placée de telle sorte que ä soit avant le b.

### 47.2.10 Internationalisation

Si vous vous reposez sur POSIX (une supposition plutôt large, en pratique ça signifie UNIX) vous pouvez lire plus à propos du système de locale POSIX dans *perllocale*. Le système de locale tente de rendre les choses un petit peu plus portable ou au moins plus pratique et facile d'accès pour des utilisateur non anglophone. Le système concerne les jeux de caractères et leur encodage, les formats des dates et des temps, entre autres choses.

### 47.2.11 Ressources Système

Si votre code est destiné à des systèmes ayant peu ou pas de système de mémoire virtuel, vous devrez alors être *particulièrement* attentif à éviter des constructions telles que:

```
NOTE: Ceci n'est plus "mauvais" en perl5.005
for (0..10000000) {} # mauvais
for (my $x = 0; $x <= 10000000; ++$x) {} # bon

@lines = <VERY_LARGE_FILE>; # mauvais

while (<FILE>) {$file .= $_} # parfois mauvais
$file = join(' ', <FILE>); # meilleur
```

Les deux dernières apparaissent moins intuitives pour la plupart des gens. La première méthode agrandi répétitivement une chaîne, tandis que la seconde alloue un large bloc de mémoire en une seule fois. Sur certain système, cette dernière méthode est plus efficace que l'autre.

### 47.2.12 Sécurité

La plupart des plateformes multi-utilisateurs fournissent des niveaux de sécurité de base reposant sur le système de fichier. Certaines ne le font pas (Malheureusement). Dès lors les notions d'identifiant utilisateur, de répertoire de "base", et même l'état d'être connecté, ne sont pas reconnus par un grand nombre de plateformes. Si vous écrivez des programmes dépendant de la sécurité, il est très utile de connaître le type de systèmes qui sera utilisé et d'écrire du code spécifiquement pour cette plateforme (ou ce type de plateforme).

### 47.2.13 Style

S'il est nécessaire d'avoir du code spécifique à une plateforme, essayez de regrouper tout ce qui est spécifique en un seul endroit, rendant le portage plus facile. Utilisez le module `Config` et la variable spéciale `$^O` pour différencier les plateformes, comme d'écrit dans `PLATEFORMES` (§47.4).

## 47.3 Testeurs du CPAN

Les modules disponibles sur le CPAN sont testés par un groupe de volontaires sur des plateformes différentes. Ces testeurs sont prévenus par mail de chaque nouveau téléchargement, et répondent sur la liste par PASS, FAIL, NA (Non applicable à cette plateforme ou UNKNOWN (inconnu), accompagné d'éventuels commentaires.

Le but de ce test est double: premièrement, aider les développeurs à supprimer des problèmes de leur code qui n'étaient pas apparus par manque de test sur d'autres plateformes; deuxièmement fournir aux utilisateurs des informations sur le fonctionnement ou non d'un module donné sur une plateforme donnée.

**Mailing list:** [cpan-testers@perl.org](mailto:cpan-testers@perl.org)

**Résultats des tests:** <http://www.connect.net/gbarr/cpan-test/>

## 47.4 PLATEFORMES

Depuis la version 5.002, Perl fournit une variable `$^O` qui indique le système d'exploitation sur lequel il a été compilé. Elle fut implémentée pour aider à accélérer du code qui autrement aurait dû utiliser `use Config`; et la valeur de `$Config{'osname'}`. Bien sûr, pour obtenir des informations détaillées sur le système, regarder `%Config` est très recommandé.

### 47.4.1 Unix

Perl fonctionne sur une grande majorité d'Unix et d'Unix-like (par exemple regardez la plupart des fichiers du répertoire `hints/` du code source). Sur la plupart de ces systèmes, la valeur de `$^O` (comme `$Config{'osname'}`) est déterminée en retirant la ponctuation et mettant en minuscule le premier champ de la chaîne retournée en tapant `uname -a` (ou une commande similaire) au prompt. Voici par exemple pour les Unix les plus populaires:

<code>uname</code>	<code>\$^O</code>	<code>\$Config{'archname'}</code>
AIX	<code>aix</code>	<code>aix</code>
FreeBSD	<code>freebsd</code>	<code>freebsd-i386</code>
Linux	<code>linux</code>	<code>i386-linux</code>
HP-UX	<code>hpux</code>	<code>PA-RISC1.1</code>
IRIX	<code>irix</code>	<code>irix</code>
OSF1	<code>dec_osf</code>	<code>alpha-dec_osf</code>
SunOS	<code>solaris</code>	<code>sun4-solaris</code>
SunOS	<code>solaris</code>	<code>i86pc-solaris</code>
SunOS4	<code>sunos</code>	<code>sun4-sunos</code>

Notez que parce que `$Config{'archname'}` peut dépendre du type d'architecture matérielle elle peut varier beaucoup, bien plus que `$^O`.



### 47.4.2 DOS dérivés

Perl a été porté sur les PC tournant sous des systèmes tels que PC-DOS, MS-DOS, OS/2, et la plupart des plateformes Windows que vous pourriez imaginer (à l'exception de Windows CE, si vous le comptiez comme tel). Les utilisateurs familiarisés du style de shell *COMMAND.COM* et/ou *CMD.EXE* doivent être conscients que chacune de ces spécifications de fichiers possède de subtiles différences:

```
$filespec0 = "c:/foo/bar/file.txt";
$filespec1 = "c:\\foo\\bar\\file.txt";
$filespec2 = 'c:\foo\bar\file.txt';
$filespec3 = 'c:\\foo\\bar\\file.txt';
```

Les appels systèmes peuvent accepter indifféremment / ou \ comme séparateur de chemin. Mais attention, beaucoup d'utilitaires en ligne de commande du type DOS traitent / comme un préfixe d'option, ils peuvent donc être perturbés par des noms de fichiers contenant /. À part lors des appels de programmes externes, / fonctionne parfaitement, et probablement mieux, car il est plus consistant avec l'usage populaire, et évite d'avoir à se souvenir ce qui doit être protégé par / et ce qui ne doit pas l'être.

Le système de fichiers FAT de DOS ne peut utiliser des noms de fichiers "8.3". Sous les systèmes de fichiers "insensibles à la casse, mais préservant la casse" HPFS (OS/2) et NTFS (NT) vous devez faire attention à la casse renvoyée par des fonctions comme `readdir` ou utilisée par des fonctions telles que `open` ou `opendir`.

DOS traite aussi quelques noms de fichiers comme étant spéciaux, tels que AUX, PRN, NUL, CON, COM1, LPT1, LPT2 etc. Malheureusement ces noms de fichiers ne fonctionneront pas même si vous y préfixez un chemin absolu. Il vaut donc mieux éviter de tels noms, si vous voulez que votre code soit portable sous DOS et ses dérivés.

Les utilisateurs de ces systèmes d'exploitations peuvent aussi vouloir utiliser des scripts tels que *pl2bat.bat* ou *pl2cmd* pour envelopper leurs scripts.

Les retours chariots (\n) sont convertis en \015\012 par `STDIO` lors de l'écriture et la lecture des fichiers. `binmode(FILEHANDLE)` conservera \n comme \n pour ce handle. Dès lors que c'est une opération sans effet sur les autres systèmes, `binmode` devrait être utilisé pour les programmes cross-plateforme qui travaillent sur des données binaires.

Voici les valeurs des variables `$^O` et `$Config{'archname'}` pour les différentes versions de DOS perl:

OS	<code>\$^O</code>	<code>\$Config{'archname'}</code>
MS-DOS	dos	
PC-DOS	dos	
OS/2	os2	
Windows 95	MSWin32	MSWin32-x86
Windows NT	MSWin32	MSWin32-x86
Windows NT	MSWin32	MSWin32-alpha
Windows NT	MSWin32	MSWin32-ppc

Consultez aussi:

**L'environnement djgpp pour DOS, <http://www.delorie.com/djgpp/>**

**L'environnement EMX pour DOS, OS/2, etc. [emx@iaehv.nl](mailto:emx@iaehv.nl), <http://www.juge.com/bbs/Hobb.19.html>**

**Instruction pour la compilation sur Win32, [perlwin32](#).**

**The ActiveState Pages, <http://www.activestate.com/>**

### 47.4.3 Mac OS

N'importe quel module utilisant la compilation XS, sont hors de portée de la plupart des gens, car MacPerl est construit en utilisant un compilateur payant (et pas qu'un peu !). Quelques modules XS pouvant fonctionner avec MacPerl sont fournis en package binaire sur le CPAN. Consultez *MacPerl: Power and Ease* et Testeurs du CPAN (§47.3) pour plus de détails.

Les répertoires sont spécifiés de la manière suivante:

volume:répertoire:fichier	chemin absolu
volume:répertoire:	chemin absolu
:répertoire:fichier	chemin relatif
:répertoire:	chemin relatif
:fichier	chemin relatif
fichier	chemin relatif

Les fichiers d'un répertoire sont stockés dans l'ordre alphabétique. Les noms de fichiers sont limités à 31 caractères différent de `.`, qui est réservé comme séparateur de chemin.

Utilisez `FSpSetFlock` et `FSpRstFlock` du module `Mac::Files` à la place de `flock`.

Les application MacPerl ne peuvent être lancées en ligne de commande; pour les programme s'attendant à recevoir des paramètres dans `@ARGV` peuvent utiliser le code suivant qui ouvre une boîte de dialogue demandant les arguments.

```
if (!@ARGV) {
 @ARGV = split /\s+/, MacPerl::Ask('Arguments?');
}
```

Si un script MacPerl est lancé grâce à du Drag'n'drop, `@ARGV` contiendra les noms de fichiers lâchés sur le script.

Les utilisateurs de Mac peuvent utiliser les programmes sous une sorte de ligne de commande grâce à MPW (Macintosh Programmer's Workshop, un environnement de développement libre d'Apple). MacPerl à d'abord été introduit comme un outil MPW, et MPW peut être utilisé comme un shell:

```
perl myscript.plx des arguments
```

ToolServer est une autre application d'Apple qui fournit un accès aux outil MPW et aux application MacPerl, permettant aux programmes MacPerl d'utiliser `system`, `'`, et `open` avec un tube.

"Mac OS" est le nom propre du système d'exploitation, mais la valeur de `$^O` est "MacOS". Pour déterminer l'architecture, la version, ou soit la version de l'application ou de l'outil MPW, regardez:

```
$is_app = $MacPerl::Version =~ /App/;
$is_tool = $MacPerl::Version =~ /MPW/;
($version) = $MacPerl::Version =~ /^(\S+)/;
$is_ppc = $MacPerl::Architecture eq 'MacPPC';
$is_68k = $MacPerl::Architecture eq 'Mac68K';
```

Mac OS X, est basé sur OpenStep de NeXT, sera capable d'exécuter MacPerl en natif (dans la boîte bleue, et même dans la boîte jaune, une fois que certains changement des appels toolbox seront faits).

Consultez aussi:

**The MacPerl Pages**, <http://www.ptf.com/macperl/>.

**The MacPerl mailing list**, [mac-perl-request@iis.ee.ethz.ch](mailto:mac-perl-request@iis.ee.ethz.ch).

#### 47.4.4 VMS

L'utilisation de Perl sous VMS est expliquée dans le `vms/perl/vms.pod` de la distribution perl. Notez que perl sur VMS peut accepter indifféremment les spécifications de fichiers de style VMS et Unix:

```
$ perl -ne "print if /perl_setup/i" SYS$LOGIN:LOGIN.COM
$ perl -ne "print if /perl_setup/i" /sys$login/login.com
```

mais pas un mélange des deux:

```
$ perl -ne "print if /perl_setup/i" sys$login:/login.com
Can't open sys$login:/login.com: file specification syntax error
```

Utiliser Perl à partir du shell Digital Command Language (DCL) nécessite souvent de d'autres marques de quotation que sous un shell Unix. Par exemple:

```
$ perl -e "print ""Hello, world.\n""
Hello, world.
```

Il y a plusieurs moyen d'inclure vos scripts perl dans un fichier DCL .COM si vous le souhaitez. Par exemple:

```
$ write sys$output "Hello from DCL!"
$ if p1 .eqs. ""
$ then perl -x 'f$environment("PROCEDURE")
$ else perl -x - 'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$ deck/dollars="__END__"
#!/usr/bin/perl

print "Hello from Perl!\n";

__END__
$ endif
```

Tenez compte du \$ ASSIGN/nolog/user SYS\$COMMAND: SYS\$INPUT si votre script perl inclus essaye de faire des choses telle que \$read = <STDIN>;.

Les noms de fichiers sont dans le format "nom.extension;version". La taille maximale est de 39 caractères pour le nom de fichier et aussi 39 pour l'extension. La version est un nombre compris entre 1 et 32767. Les caractères valides sont /[A-Z0-9\$\_-]/. Le système de fichier RMS de VMS est insensible à la casse et ne la préserve pas. readdir renvoie un nom de fichier en minuscule, mais l'ouverture reste insensible à la case. Les fichiers sans extensions sont terminés par un point, donc faire un readdir avec un fichier nommé A.;5 retournera a. (ce fichier peut être ouvert par open(FH, 'A')).

RMS a une limitation à huit niveau de profondeur pour les répertoires à partir de n'importe quel racine logique ( autorisant un maximum de 16 niveau) avant VMS 7.2. Dès lors PERL\_ROOT: [LIB.2.3.4.5.6.7.8] est un répertoire valide mais PERL\_ROOT: [LIB.2.3.4.5.6.7.8.9] non. Les auteurs de *Makefile.PL* peuvent l'avoir pris en compte, mais il peuvent au moins référencer le dernier par /PERL\_ROOT/lib/2/3/4/5/6/7/8/.

Le module VMS::Filespec, qui est installé lors de la compilation sur VMS, est un module en Perl pure qui peut être installé facilement sur des plateformes non-VMS et peut être utile pour les conversion à partir et vers des formats natifs RMS.

La valeur de \n dépend du type de fichier qui est ouvert. Ça peut être \015, \012, \015\012, ou rien du tout. Lire d'un fichier traduit les retour chariots en \012, à moins que binmode soit exécuté sur ce handle de fichier, comme pour les perls DOS.

La pile TCP/IP est optionnel sur VMS, les routines socket ne sont donc pas forcément installés. Les sockets UDP peuvent ne pas être supportés.

La valeur de \$^O sur OpenVMS est "VMS". Pour déterminer l'architecture sur laquelle le programme est exécuté sans utiliser %Config vous pouvez examiner le contenu du tableau @INC:

```
if (grep(/VMS_AXP/, @INC)) {
 print "I'm on Alpha!\n";
} elsif (grep(/VMS_VAX/, @INC)) {
 print "I'm on VAX!\n";
} else {
 print "I'm not so sure about where $^O is...\n";
}
```

Consultez aussi:

*perlvms.pod*

**vmsperl list, vmsperl-request@newman.upenn.edu**

Précisez SUBSCRIBE VMSPERL dans le corps du message.

**vmsperl on the web, <http://www.sidhe.org/vmsperl/index.html>**

### 47.4.5 Plateformes EBCDIC

Les versions récentes de Perl ont été portées sur des plateformes telles que OS/400 sur les minicomputers AS/400 ainsi que OS/390 pour Mainframes IBM. De tels ordinateurs utilise le jeu de caractère EBCDIC en interne (habituellement le jeu de caractère ID 00819 sur OS/400 et IBM-1047 sur OS/390). Notez que sur les mainframe perl fonctionne actuellement sous le "Unix system services for OS/390" (connu précédemment sous le nom OpenEdition).

La version actuelle, la R2.5 de USS pour OS/390, de ce sous-système Unix ne supporte pas l'astuce du #! pour l'invocation des scripts. Mais, sur le OS/390 les script peuvent utiliser un en-tête similaire à ceci:

```
: # use perl
 eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
 if 0;
#!/usr/local/bin/perl # juste un commentaire

print "Hello from perl!\n";
```

Sur ces plateformes, gardez à l'esprit que le jeu de caractère EBCDIC peut avoir un effet sur certaines fonctions perl (telles que `chr`, `pack`, `print`, `printf`, `ord`, `sort`, `sprintf`, `unpack`), aussi bien que sur les modification binaire à l'aide de constante ASCII en utilisant des opérateurs comme `^`, `&` et `|`, sans oublier de mentionner l'interface avec des ordinateurs ASCII en utilisant des sockets. (cf Les retours chariots (§47.2.1)).

Heureusement, la plupart des serveurs web pour mainframe convertissent correctement les `\n` de l'instruction suivante en leurs équivalent ASCII (notez que `\r` est le même sous Unix et OS/390):

```
print "Content-type: text/html\r\n\r\n";
```

La valeur de `$^O` sur OS/390 est "os390".

Voici des moyens simple de déterminé si le programme tourne sur une plateforme EBCDIC :

```
if ("\t" eq "\05") { print "EBCDIC may be spoken here!\n"; }

if (ord('A') == 193) { print "EBCDIC may be spoken here!\n"; }

if (chr(169) eq 'z') { print "EBCDIC may be spoken here!\n"; }
```

Notez que vous ne devez pas vous reposer sur les caractères de ponctuation car leurs codes EBCDIC peut changer de page de code en page de code (et une fois que votre module ou script est connu comme fonctionnant avec EBCDIC, Il y aura toujours quelqu'un qui voudra qu'il fonctionne avec tous les jeux de caractères EBCDIC).

Consultez aussi:

#### perl-mvs list

La liste `perl-mvs@perl.org` est pour les discussions sur les problèmes de portage ainsi que les problèmes généraux de l'utilisation des Perls. EBCDIC. Envoyer un message contenant "subscribe perl-mvs" à `majordomo@perl.org`.

**AS/400 Perl information <http://as400.rochester.ibm.com/>**

### 47.4.6 Acorn RISC OS

Comme les Acorns utilise l'ASCII avec les retour chariots (`\n`) des fichiers texte égal à `\012` comme sous Unix et que l'émulation des noms de fichier Unix est validée par défaut, il y a de grande chance pour que la plupart des scripts simples fonctionne sans modification. Le système de fichier natif est modulaire, et ces modules peuvent être insensible ou sensible à la casse, et conservent en général la casse des noms de fichier. Certains de ces modules ont des limites sur la taille des noms de fichier et ont alors tendance à tronquer les noms des fichiers et des répertoires pour qu'ils ne dépassent pas cette limite - les scripts doivent être savoir que le module par défaut est limité à **10** caractères et à un maximum de **77** éléments dans un répertoire, mais que la plupart n'imposent pas de telles limitations.

Les noms de fichier natifs sont de la forme

```
SystèmeDeFichier#ChampSpécial::NomDisque.$Répertoire.Répertoire.Fichier
```

où

```

ChampSpécial n'est en général pas présent mais peut contenir . et $.
SystèmeDeFichier =~ m|[A-Za-z0-9_]|
NomDisque =~ m|[A-Za-z0-9_/]|
$ representes le répertoire racine
. est le séparateur de chemin
@ est le répertoire courant (par Système de fichier mais global à la machine)
^ est le répertoire parent
Répertoire and Fichier =~ m|^\0- "\.\$%\&:\@\^\|\177]+|

```

La conversion par défaut des noms de fichier est à peu près `tr|/|.|/;`

Notez que `"ADFS::HardDisc$.File"` ne `'ADFS::HardDisc$.File'` et que la deuxième étape de l'interprétation de `$` dans les expression régulière risque d'échouer sur le `$`. si le script ne fait pas attention.

Les chemins logiques spécifiés par des variables d'environnement contenant des listes de recherche séparées par des virgules sont aussi autorisées, de même que `System:Modules` est un nom de fichier valide, et que les système préfixera `Modules` par toutes les section de `System$Path` jusqu'à ce qu'un nom ainsi conçu pointe sur un objet sur disque. Écrire un nouveau fichier `System:Modules` ne sera autorisé que si `System$Path` un seul chemin. Le système de fichier converti aussi les variables systèmes en nom de fichier si elles sont entourées de signe `<>`, donc `<System$Dir>.Modules` recherche le fichier `$ENV{'System$Dir'}. 'Modules'`. L'explication évidente est que **les noms de fichiers absolus peuvent commencer par <>** et doivent être protégé lorsque l'on utilise `open` pour la lecture de donnée.

Comme `.` est le séparateur de chemin et que les noms de fichiers ne peuvent pas être présumé être unique après le dixième caractère, Acorn a implémenté le compilateur C de manière à retirer les extensions `.c .h .s` et `.o` des noms de fichiers spécifiés dans le code source et de stocker les fichiers respectifs dans des sous-répertoire au nom correspondant à l'extension. En voici quelques exemples:

```

foo.h h.foo
C:foo.h C:h.foo (Variable de chemin logique)
sys/os.h sys.h.os (Le compilateur C baragouine de l'Unix)
10charname.c c.10charname
10charname.o o.10charname
11charname_.c c.11charname (présume que le système de fichier tronque à 10)

```

La librairie d'émulation Unix convertit les noms de fichier en natif en présument que ce type de conversion est nécessaire, et permet à l'utilisateur de définir sa liste d'extension pour lesquelles une telle conversion est nécessaire. Ça peut sembler transparent, mais considérez que `foo/bar/baz.h` et `foo/bar/h/baz` se traduisent tous les deux par `foo.bar.h.baz`, et que `readdir` et `glob` ne peuvent et n'essayeront pas d'émuler la conversion inverse. Les autres `.` dans les noms de fichiers seront convertis en `/`.

Comme précisé précédemment l'environnement accessible par `%ENV` est global, et la convention est qu'une variable spécifique à un programme soit de la forme `Programme$Nom`. Chaque module de systèmes de fichiers maintiennent un répertoire courant et celui du module courant est le répertoire courant **global**. En conséquence, les scripts sociaux ne change pas le répertoire courant, mais repose sur des noms de fichiers complets, et les scripts (et les Makefiles) ne peuvent supposer que le faite de créer un processus fils ne va pas modifier le répertoire courant de son père (et de tous les autres processus en général).

Les handle de fichier natifs du système d'exploitation sont globaux et sont alloués en descendant depuis 255, 0 étant réservé à la librairie d'émulation Unix. Par conséquence, ne vous reposez pas sur le passage de `STDIN`, `STDOUT`, ou `STDERR` à vos enfants.

Le désir des utilisateur d'exprimer des noms de fichier de la forme `<Foo$Dir>.Bar` sur la ligne de commande sans être protégés pose un problème: La sortie résultant de la commande " doit jouer aux devinettes. Il présume que la chaîne `<[^<>]+\$[^<>>` est une référence vers une variable d'environnement, alors que tout le reste impliquant `<` ou `>` est une redirection, ce qui est 99% du temps exact. Bien sûr un problème récurrent est que les scripts ne peuvent compter sur le fait qu'un quelconque outil Unix soit disponible, et que s'il l'est, il utilise les mêmes arguments que la version Unix.

Les extensions et XS sont, en théorie, compilable par tous en utilisant des logiciels libres. En pratique, beaucoup ne le font pas, les utilisateurs des plateformes Acorn étant habitués aux distributions binaires. `MakeMaker` fonctionne, mais aucun `make` n'arrivera à traiter un Makefile créé par `MakeMaker`; et même si ceci est un jour corrigé, le manque d'un shell de type Unix pourra poser des problèmes avec les règles du makefile, particulièrement les lignes de la forme `cd sdbm && make all`, et tout ce qui utilise les protections.

"RISC OS" est le nom propre du système d'exploitation, mais la valeur de `$^O` est "riscos" (parce que nous n'ayons pas crier).

Consultez aussi:

**perl list**

### 47.4.7 Autres perls

Perl a été porté sur un très grand nombre de plateformes ne correspondant à aucune des catégories ci-dessus. Certaines, telles que AmigaOS, BeOS, QNX, et Plan 9, ont été intégrées dans le code source Perl standard. Vous pourriez avoir à consulter le répertoire *ports/* du CPAN pour des informations, et même pour des binaires pour: aos, atari, lynxos, riscos, Tandem Guardian, vos, *etc.* (oui, nous savons que certains de ces OS peuvent correspondre à la catégorie Unix, mais ils n'en sont pas complètement standard.)

Consultez aussi:

**Atari, Guido Flohr's page** <http://stud.uni-sb.de/~guf10000/>

**HP 300 MPE/iX** <http://www.cccd.edu/~markb/perl.ix.html>

**Novell Netware**

Un PERL.NLM libre basé sur perl5 est disponible pour Novell Netware à partir de <http://www.novell.com/>

## 47.5 IMPLÉMENTATION DES FONCTIONS

La liste ci-dessous contient les fonctions qui ne sont pas implémentées ou différemment sur des plateformes diverses. Chaque description est suivie de la liste des plateformes auxquelles elle s'applique.

La liste peut parfaitement être incomplète voire même fautive. En cas de doute, consultez le README spécifique à la plateforme de la distribution des sources perl, et d'autres documents d'un portage donné.

Faites attention, que même les Unix connaissent des différences.

Pour beaucoup de fonctions, vous pouvez aussi interroger `%Config`, exporté par défaut de `Config.pm`. Par exemple, pour vérifier que la plateforme connaît l'appel système `lstat`, consultez `$Config{'d_lstat'}`. Lisez `Config.pm` pour une description complète des variables disponibles.

### 47.5.1 Liste Alphabétique des Fonctions Perl

#### -X FILEHANDLE

#### -X EXPR

#### -X

`-r`, `-w`, et `-x` ont un sens très limité; les applications et les répertoires sont exécutables, et il n'existe pas uid/gid (Mac OS)

`-r`, `-w`, `-x`, et `-o` indiquent si un fichier est oui ou non accessible qui peut ne pas refléter les protections de fichiers basées sur UIC. (VMS)

`-s` retourne la taille des données et non pas la taille totale des données et des ressources. (Mac OS)

`-s` par nom sur un fichier ouvert retournera la place réservée sur disque plutôt que la taille actuelle. `-s` sur un handle retournera lui la taille courante (RISC OS)

`-R`, `-W`, `-X`, `-O` ne peuvent être différenciés de `-r`, `-w`, `-x`, `-o`. (Mac OS, Win32, VMS, RISC OS)

`-b`, `-c`, `-k`, `-g`, `-p`, `-u`, `-A` ne sont pas implémentés. (Mac OS)

`-g`, `-k`, `-l`, `-p`, `-u`, `-A` n'ont pas beaucoup de sens. (Win32, VMS, RISC OS)

`-d` est vrai si on lui passe une spécification de périphérique sans un répertoire explicite. (VMS)

`-T` et `-B` sont implémentés, mais peuvent mal classer les fichiers texte Mac contenant des caractères étrangers; ça arrive sur toutes les plateformes, mais affecte plus souvent Mac OS. (Mac OS)

`-x` (ou `-X`) détermine si le fichier possède une extension exécutable `-S` n'a pas de sens. (Win32)

`-x` (ou `-X`) détermine si un fichier est du type exécutable. (RISC OS)

#### binmode FILEHANDLE

Sans intérêt. (Mac OS, RISC OS)

Rouvre le fichier et réinitialise les pointeurs; si la fonction échoue, le handle de fichier peut être fermé, ou le pointeur peut être à une position différente. (VMS)

La valeur retournée par `tell` peut être différente après l'appel, et le handle peut être vidé. (Win32)

#### chmod LIST

Son sens est limité. Désactiver/activer la permission en écriture est remplacé par la désactivation/activation d'un verrou sur le fichier. (Mac OS)

Ne peut modifier que les permissions lecture/écriture pour le "propriétaire", les droits du "groupe" et des "autres" n'ont pas de signification. (Win32)

Ne peut modifier que les permissions lecture/écriture pour le "propriétaire", et les "autres". (RISC OS)

**chown LIST**

Non implémenté. (Mac OS, Win32, Plan9, RISC OS)

Ne fait rien, mais ne renvoie pas d'erreur. (Win32)

**chroot FILENAME****chroot**

Non implémenté. (Mac OS, Win32, VMS, Plan9, RISC OS)

**crypt PLAINTEXT,SALT**

Peut ne pas être disponible si la librairie ou le source n'était pas fourni lors de la compilation de perl. (Win32)

**dbmclose HASH**

Non implémenté. (VMS, Plan9)

**dbmopen HASH,DBNAME,MODE**

Non implémenté. (VMS, Plan9)

**dump LABEL**

Pas utile. (Mac OS, RISC OS)

Non implémenté. (Win32)

Invoke le debogeur VMS . (VMS)

**exec LIST**

Non implémenté. (Mac OS)

**fcntl FILEHANDLE,FUNCTION,SCALAR**

Non implémenté. (Win32, VMS)

**flock FILEHANDLE,OPERATION**

Non implémenté (Mac OS, VMS, RISC OS).

Disponible uniquement sous Windows NT (pas sous Windows 95). (Win32)

**fork**

Non implémenté. (Mac OS, Win32, AmigaOS, RISC OS)

**getlogin**

Non implémenté. (Mac OS, RISC OS)

**getpgrp PID**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**getppid**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**getpriority WHICH,WHO**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**getpwnam NAME**

Non implémenté. (Mac OS, Win32)

Pas utile. (RISC OS)

**getgrnam NAME**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**getnetbyname NAME**

Non implémenté. (Mac OS, Win32, Plan9)

**getpwuid UID**

Non implémenté. (Mac OS, Win32)

Pas utile. (RISC OS)

**getgrgid GID**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**getnetbyaddr ADDR,ADDRTYPE**

Non implémenté. (Mac OS, Win32, Plan9)

**getprotobynumber NUMBER**

Non implémenté. (Mac OS)

**getservbyport PORT,PROTO**

Non implémenté. (Mac OS)

**getpwent**

Non implémenté. (Mac OS, Win32)

**getgrent**

Non implémenté. (Mac OS, Win32, VMS)

**gethostent**

Non implémenté. (Mac OS, Win32)

**getnetent**

Non implémenté. (Mac OS, Win32, Plan9)

**getprotoent**

Non implémenté. (Mac OS, Win32, Plan9)

**getservent**

Non implémenté. (Win32, Plan9)

**setpwent**

Non implémenté. (Mac OS, Win32, RISC OS)

**setgrent**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**sethostent STAYOPEN**

Non implémenté. (Mac OS, Win32, Plan9, RISC OS)

**setnetent STAYOPEN**

Non implémenté. (Mac OS, Win32, Plan9, RISC OS)

**setprotoent STAYOPEN**

Non implémenté. (Mac OS, Win32, Plan9, RISC OS)

**setservent STAYOPEN**

Non implémenté. (Plan9, Win32, RISC OS)

**endpwent**

Non implémenté. (Mac OS, Win32)

**endgrent**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**endhostent**

Non implémenté. (Mac OS, Win32)

**endnetent**

Non implémenté. (Mac OS, Win32, Plan9)

**endprotoent**

Non implémenté. (Mac OS, Win32, Plan9)

**endservent**

Non implémenté. (Plan9, Win32)

**getsockopt SOCKET,LEVEL,OPTNAME**

Non implémenté. (Mac OS, Plan9)

**glob EXPR****glob**

La complétion est interne, mais seuls le méta-caractères \* et ? sont supportés. (Mac OS)

Fonctionnalité dépendant d'un programme externe perlglob.exe ou perlglob.bat peut être surpassé par quelque chose comme File::DosGlob, ce qui est recommandé. (Win32)

La complétion est interne, mais seuls le méta-caractères \* et ? sont supportés. Elle dépends d'appels système, qui peuvent retourner les noms de fichier dans n'importe quel ordre. Comme la plupart des systèmes de fichier sont insensible à la casse, même les noms "triés" ne le seront pas dans un ordre dépendant de la casse. (RISC OS)

**ioctl FILEHANDLE,FUNCTION,SCALAR**

Non implémenté. (VMS)

Disponible seulement pour les handles de socket, et ne fait que ce que l'appel ioctlsocket() de l'API Winsock fait. (Win32)

Disponible seulement pour les handles de socket. (RISC OS)



**kill LIST**

Non implémenté, même pas utile pour la vérification de sécurité. (Mac OS, RISC OS)

Disponible uniquement pour les handles de processus retourné par `system(1, ...)`. (Win32)

**link OLDFILE,NEWFILE**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**lstat FILEHANDLE****lstat EXPR****lstat**

Non implémenté. (VMS, RISC OS)

La valeur de retour peut être farfelu. (Win32)

**msgctl ID,CMD,ARG****msgget KEY,FLAGS****msgsnd ID,MSG,FLAGS****msgrcv ID,VAR,SIZE,TYPE,FLAGS**

Non implémenté. (Mac OS, Win32, VMS, Plan9, RISC OS)

**open FILEHANDLE,EXPR****open FILEHANDLE**

Les variantes | ne sont supportés que si ToolServer est installé. (Mac OS)

L'ouverture vers |- et -| ne sont pas supportés. (Mac OS, Win32, RISC OS)

**pipe READHANDLE,WRITEHANDLE**

Non implémenté. (Mac OS)

**readlink EXPR****readlink**

Non implémenté. (Win32, VMS, RISC OS)

**select RBITS,WBITS,EBITS,TIMEOUT**

Implémentés seulement sur les sockets. (Win32)

Fiables que sur les sockets. (RISC OS)

**semctl ID,SEMNUM,CMD,ARG****semget KEY,NSEMS,FLAGS****semop KEY,OPSTRING**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**setpgrp PID,PGRP**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**setpriority WHICH,WHO,PRIORITY**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL**

Non implémenté. (Mac OS, Plan9)

**shmctl ID,CMD,ARG****shmget KEY,SIZE,FLAGS****shmread ID,VAR,POS,SIZE****shmwrite ID,STRING,POS,SIZE**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**stat FILEHANDLE****stat EXPR****stat**

`mtime` et `atime` représente la même chose, et `ctime` est la date de création au lieu de la date du dernier changement de l'inode. (Mac OS)

`device` et `inode` n'ont aucun sens. (Win32)

`device` et `inode` ne sont pas forcément fiables. (VMS)

`mtime`, `atime` et `ctime` correspondent à la date de dernière modification. `device` et `inode` ne sont pas forcément fiables. (RISC OS)

**symlink OLDFILE,NEWFILE**

Non implémenté. (Win32, VMS, RISC OS)

**syscall LIST**

Non implémenté. (Mac OS, Win32, VMS, RISC OS)

**sysopen FILEHANDLE,FILENAME,MODE,PERMS**

Les valeurs traditionnelles des modes "0", "1", et "2" sont implémentés par des valeurs numériques différentes sur quelques systèmes. Les drapeaux exportés par `Fcntl` (`O_RDONLY`, `O_WRONLY`, `O_RDWR`) devraient fonctionner sur tout les systèmes. (Mac OS, OS/390)

**system LIST**

Implémenté que si ToolServer est installé. (Mac OS)

Comme optimisation, peut ne pas faire appel au shell de commande spécifié par `$ENV{PERLSHELL}`. `system(1, @args)` crée un processus externe et renvoie immédiatement le handle de processus, sans attendre qu'il se termine. La valeur de retour peut donc être utilisée par `wait` ou `waitpid`. (Win32)

Il n'y a pas de shell pour traiter les méta-caractères, et le standard natif est de passer des ligne de commandes terminées par `"\n"`, `"\r"` ou `"\0"` au processus créé. Les redirections tels que `> foo` sont réalisées par la librairie d'exécution du processus créé. `system list` fait appel à la fonction `exec` de la librairie d'émulation Unix, qui tente de fournir une émulation de `stdin`, `stdout`, `stderr` en forçant le père, présumant que le programme fils utilise une version compatible de la librairie d'émulation. `scalar` fait appel à la ligne de commande native et une telle émulation n'est pas utilisé. Ce fonctionnement va varier. (RISC OS)

**times**

Seule la première entrée est différente de zéro. (Mac OS)

Les temps "cumulés" sont faux. Sur tout les systèmes autres que Windows NT, le temps "système" est faux, et le temps "utilisateur" est actuellement la date retourné par la fonction `clock()` de la librairie C. (Win32)

Pas utile. (RISC OS)

**truncate FILEHANDLE,LENGTH****truncate EXPR,LENGTH**

Non implémenté. (VMS)

**umask EXPR****umask**

Retourne undef lorsqu'elle n'est pas disponible depuis la version 5.005.

**utime LIST**

Seule la date de modification est mise à jour. (Mac OS, VMS, RISC OS)

Peut ne pas fonctionné comme attendu. Le comportement dépend l'implémentation de la fonction `utime()` de la librairie C, et du type de système de fichier utilisé. Le système FAT typiquement ne supporte pas le champs "date d'accès", et peut limiter les marques de temps à une granularité de deux secondes. (Win32)

**wait****waitpid PID,FLAGS**

Non implémenté. (Mac OS)

Disponible uniquement pour les handles de processus retourné par `system(1, ...)`. (Win32)

Pas utile. (RISC OS)

## 47.6 AUTEURS / CONTRIBUTEURS

Abigail <abigail@fnx.com>, Charles Bailey <bailey@genetics.upenn.edu>, Graham Barr <gbarr@pobox.com>, Tom Christiansen <tchrist@perl.com>, Nicholas Clark <Nicholas.Clark@liverpool.ac.uk>, Andy Dougherty <doughera@lafcol.lafayette.edu>, Dominic Dunlop <domo@vo.lu>, M.J.T. Guy <mjtg@cus.cam.ac.uk>, Luther Huffman <lutherh@stratcom.com>, Nick Ing-Simmons <nick@ni-s.u-net.com>, Andreas J. König <koenig@kulturbox.de>, Andrew M. Langmead <aml@world.std.com>, Paul Moore <Paul.Moore@uk.origin-it.com>, Chris Nandor <pudge@pobox.com>, Matthias Neeracher <neeri@iis.ee.ethz.ch>, Gary Ng <71564.1743@CompuServe.COM>, Tom Phoenix <rootbeer@teleport.com>, Peter Prymmer <pvhp@forte.com>, Hugo van der Sanden <hv@crypt0.demon.co.uk>, Gurusamy Sarathy <gsar@umich.edu>, Paul J. Schinder <schinder@pobox.com>, Dan Sugalski <sugalskd@ous.edu>, Nathan Torkington <gnat@frii.com>.

Ce document est maintenu par Chris Nandor.

## **47.7 VERSION**

Version 1.34, last modified 07 August 1998.

## **47.8 TRADUCTION**

### **47.8.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **47.8.2 Traducteur**

Marc Carmier <carmier@immortels.frmug.org>.

### **47.8.3 Relecture**

Personne pour l'instant.

# Chapitre 48

## perllocale

Gestion des "locale" en Perl (internationalisation et localisation)

### 48.1 DESCRIPTION

Perl supporte pour les données, des notions spécifiques à la langue telles que "Est-ce une lettre", "quel est l'équivalent en majuscule de cette lettre" et "laquelle de ces lettres est la première". Ce sont des sujets très importants spécialement pour les langues autres que l'anglais – mais aussi pour l'anglais : il serait naïf d'imaginer que A-Za-z décrit toutes les "lettres" nécessaires à l'écriture de l'anglais. Perl sait aussi qu'un autre caractère que '.' peut être utilisé comme séparateur décimal et que la représentation d'une date est liée à la langue. La tâche consistant à rendre une application capable de gérer les préférences de l'utilisateur dans ce domaine s'appelle l'**internationalisation** (parfois abrégé en **i18n**). Spécifier à une telle application l'ensemble de ces préférences pour une langue particulière s'appelle la **localisation (l10n)**.

Perl peut comprendre les données spécifiques à une langue via la méthode standard (ISO C, XPG4, POSIX 1.c) appelée système de "locale". Le système de "locale" est contrôlé par une application en utilisant un pragma (une directive), un appel de fonction et plusieurs variables d'environnement.

**Note** : cette fonctionnalité est nouvelle dans Perl 5.004 et ne doit pas s'appliquer à moins qu'une application le demande explicitement – voir Compatibilité (§48.7.1). La seule exception est write() qui maintenant utilise **toujours** le "locale" courant – voir NOTES (§48.7).

### 48.2 PRÉ-REQUIS POUR L'UTILISATION DES "LOCALE"

Pour qu'une application Perl comprenne et présente vos données correctement en fonction du "locale" de votre choix, **toutes** les conditions suivantes doivent être remplies :

- **Votre système d'exploitation doit supporter le système de "locale"**. Si c'est le cas, vous devriez trouver la fonction setlocale() dans la documentation sa bibliothèque C.
- **Les définitions des "locale" que vous voulez utiliser doivent être installées**. Vous ou votre administrateur système devez être sûr que c'est bien le cas. Les "locale" disponibles, l'endroit où ils sont stockés et la manière dont ils sont installés varient d'un système à l'autre. Certains systèmes ne fournissent que très peu de "locale" installés une bonne fois pour toutes et il n'est pas possible d'en ajouter. D'autres autorisent l'ajout de "locale" en conserve fournis par le fournisseur du système. D'autres encore vous autorisent à créer et à définir vos propres "locale". Lisez la documentation de votre système pour d'autres informations.
- **Perl doit savoir que le système de "locale" est supporté**. Si c'est le cas, perl -V:d\_setlocale doit dire que la valeur de d\_setlocale est define.

Si vous voulez qu'une application Perl traite et présente les données en fonction d'un "locale" particulier, le code de l'application devrait inclure la directive use locale (voir La directive locale) là où c'est approprié et **au moins** l'une des conditions suivantes devraient être remplies :

- **les variables d'environnement (voir ENVIRONNEMENT (§48.6)) qui déterminent le "locale" doivent être correctement positionnées** au moment du lancement de l'application soit par vous-même soit par la configuration de votre compte système.
- **L'application doit positionné son propre "locale"** en utilisant la méthode décrite dans La fonction setlocale.

## 48.3 UTILISATION DES "LOCALE"

### 48.3.1 La directive locale

Par défaut, Perl ignore le "locale" courant. La directive `use locale` demande à Perl de prendre en compte le "locale" courant pour certaines opérations :

- **Les opérateurs de comparaisons** (`lt`, `le`, `cmp`, `ge` et `gt`) et les fonctions POSIX de tri `strcoll()` et `strxfrm()` utilisent `LC_COLLATE`. `sort()` est aussi affecté si il est utilisé sans une fonction de comparaison explicite puisqu'il utilise `cmp` par défaut.
- Note:** `eq` et `ne` ne sont pas affectés par le "locale" : ils comparent toujours leurs deux opérandes octet par octet. De plus, si `cmp` trouve que ses deux opérandes sont égaux selon l'ordre spécifié par le "locale" courant, il lance aussi une comparaison octet par octet et retourne `0` (égal) uniquement si les deux opérandes sont identiques bit à bit. Si vous voulez vraiment savoir si deux chaînes - que `eq` et `cmp` considèrent comme différentes - sont égales selon l'ordre de tri du "locale", lisez Catégorie `LC_COLLATE` : tri.
- **Les expressions rationnelles et les fonctions de modification de la casse** (`uc()`, `lc()`, `ucfirst()` et `lcfirst()`) utilisent `LC_CTYPE`.
- **Les fonctions de formatage** (`printf()`, `sprintf()` et `write()`) utilisent `LC_NUMERIC`.
- **La fonction POSIX de formatage de dates** (`strftime()`) utilise `LC_TIME`.

`LC_COLLATE`, `LC_CTYPE` et autres sont présentés plus complètement dans CATÉGORIES DE "LOCALE".

Le comportement par défaut est rétabli par la directive `no locale` ou lorsqu'on sort du bloc englobant la directive `use locale`.

Une chaîne résultat de n'importe quelle opération qui utilise les informations du "locale" est souillée (tainted) puisqu'il vaut mieux ne pas faire confiance au "locale" courant. Voir SÉCURITÉ (§48.5).

### 48.3.2 La fonction setlocale

Vous pouvez changer de "locale" aussi souvent que nécessaire lors de l'exécution grâce à la fonction POSIX::`setlocale()` :

```
Cette fonctionnalité n'existait pas avant Perl 5.004
require 5.004;

Import des outils de manipulation du "locale"
depuis le module POSIX.
Cette exemple utilise : setlocale -- l'appel de fonction
LC_CTYPE -- expliqué plus bas
use POSIX qw(locale_h);

demande et sauvegarde le "locale" initial
$sold_locale = setlocale(LC_CTYPE);

setlocale(LC_CTYPE, "fr_CA.ISO8859-1");
LC_CTYPE est maintenant dans le
"locale" "French, Canada, codeset ISO 8859-1"

setlocale(LC_CTYPE, "");
LC_CTYPE revient à sa valeur par défaut définie par
les variables d'environnement LC_ALL/LC_CTYPE/LANG
Voir plus bas pour la documentation

restaure le "locale" initial
setlocale(LC_CTYPE, $sold_locale);
```

Le premier argument de `setlocale()` donne la **catégorie**, le seconde le **locale**. La catégorie indique à quel aspect du traitement des données vous voulez appliquer les règles spécifiques au "locale". Les noms des catégories sont présentés dans CATÉGORIES DE "LOCALE" et ENVIRONNEMENT (§48.6). Le "locale" est le nom d'une collection d'information correspondant à une combinaison particulière de langues, de pays ou territoires et de codage. Petit détail sur le nommage des "locale" : tous les systèmes ne nomment pas les "locale" comme dans les exemples donnés.

Si le second argument n'est pas fourni et que la catégorie est autre chose que LC\_ALL, la fonction retourne une chaîne indiquant le "locale" courant pour cette catégorie. Vous pouvez utiliser cette valeur comme second argument d'un appel à `setlocale()`.

Si le second argument n'est pas fourni et que la catégorie est LC\_ALL, le résultat est dépendant de l'implémentation. Cela peut être une chaîne concaténant tous les noms des "locale" (avec un séparateur dépendant de l'implémentation) ou un seul nom de "locale". Consultez `setlocale(3)` pour plus de détails.

Si un second argument est fourni et si il correspond à un "locale" valide, le "locale" pour cette catégorie est positionné à cette valeur et la fonction retourne la valeur du "locale" devenu le "locale" courant. Vous pouvez alors l'utiliser dans un autre appel à `setlocale()`. (Quelques implémentations retournent une valeur différente de la valeur que vous avez fournie comme second argument – pensez-y comme si vos valeurs étaient des alias).

Comme dans les exemples présentés, si le second argument est une chaîne vide, le "locale" de la catégorie revient à sa valeur par défaut spécifiée par les variables d'environnement correspondantes. Généralement, cela vous ramènera aux valeurs par défaut lorsque Perl a démarré : les changements de l'environnement faits par votre application après le démarrage peuvent ou non être pris en compte. Cela dépend de la bibliothèque C de votre système.

Si le second argument ne correspond à aucun "locale" valide, le "locale" pour cette catégorie n'est pas modifié et la fonction retourne `undef`.

Pour de plus amples informations concernant les catégories, consultez `setlocale(3)`.

### 48.3.3 Trouver les "locale"

Pour connaître les "locale" disponibles sur votre système, consultez `setlocale(3)` pour savoir où réside la liste des "locale" disponibles (cherchez dans la section *SEE ALSO* ou *VOIR AUSSI*). Si cela échoue, essayez les lignes de commandes suivantes :

```
locale -a
nlsinfo
ls /usr/lib/nls/loc
ls /usr/lib/locale
ls /usr/lib/nls
ls /usr/share/locale
```

et regardez si elles listent quelque chose qui ressemble à :

en_US.ISO8859-1	de_DE.ISO8859-1	ru_RU.ISO8859-5
en_US.iso88591	de_DE.iso88591	ru_RU.iso88595
en_US	de_DE	ru_RU
en	de	ru
english	german	russian
english.iso88591	german.iso88591	russian.iso88595
english.roman8		russian.koi8r

Malheureusement, bien que l'interface d'appel à `setlocale()` ait été standardisée, les noms des "locale" et les répertoires où la configuration réside ne le sont pas. La forme basique du nom est *langue\_territoire.codage* mais les dernières parties après la langue ne sont pas toujours présentes. La *langue* et le *pays* sont habituellement sous la forme proposée par les normes **ISO 3166** et **ISO 639**, les noms de pays et de langues abrégés sur deux lettres. La partie *codage* mentionne parfois un jeu de caractères **ISO 8859**. Par exemple, ISO 8859-1 est parfois appelé « jeu de caractères de l'Europe de l'Ouest » et peut être utilisé pour encoder la plupart des langues de l'Europe de l'Ouest. Même là, il y a plusieurs façons d'écrire le nom de ce standard. Lamentable.

Deux "locale" méritent une mention particulière : "C" et "POSIX". Pour l'instant, ce sont effectivement les mêmes "locale" : la seule différence est que l'un est défini par le standard C et l'autre par le standard POSIX. Ils définissent le **locale par défaut** que tous les programmes utilisent au démarrage en l'absence d'informations de "locale" dans leur environnement. Leur langue est l'anglais (américain) et leur jeu de caractères est l'ASCII.

**Note** : tous les systèmes n'ont pas le "locale" POSIX (tous les systèmes ne sont pas conformes POSIX). Donc utilisez "C" à chaque fois que vous avez besoin de spécifier le "locale" par défaut.

### 48.3.4 PROBLÈMES DE "LOCALE"

Vous pouvez obtenir le message d'avertissement suivant au démarrage de Perl :

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
 LC_ALL = "En_US",
 LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Cela signifie que vos réglages de "locale" ont LC\_ALL positionné à "En\_US" et que LANG existe mais n'a pas de valeur. Perl essaye de vous croire mais il ne le peut pas. À la place, Perl y renonce et retombe sur le "locale" "C", la "locale" par défaut qui est supposé fonctionner en toutes circonstances. Cela signifie que vos réglages de "locale" sont mauvais. Soit ils mentionnent des "locale" dont votre système n'a jamais entendu parler, soit l'installation des "locale" est mal faite sur votre système (par exemple, certains fichiers systèmes sont défectueux ou manquants). Il y a des solutions rapides et temporaires à ces problèmes mais aussi des solutions plus minutieuses et définitive.

### 48.3.5 Résolution temporaire des problèmes de "locale"

Les deux solutions les plus rapides sont soit de rendre Perl silencieux sur d'éventuelles inconsistances des "locale" soit de faire tourner Perl avec le "locale" par défaut ("C").

Les plaintes de Perl au sujet des problèmes de "locale" peuvent être supprimées en positionnant la variable d'environnement PERL\_BADLANG à la valeur zéro (par exemple "0"). Cette méthode ne fait que cacher le problème : vous demandez à Perl de se taire même si il voit que quelque chose ne va pas. Ne soyez donc pas surpris, plus tard, lorsque vous constaterez des dysfonctionnements de choses dépendant des "locale".

Perl peut tourner avec le "locale" "C" par défaut en positionnant la variable d'environnement LC\_ALL à "C". Cette méthode est peut-être un peu plus civilisée que l'approche par PERL\_BADLANG mais la valeur de LC\_ALL (ou de tout autre variable liée au "locale") peut affecter aussi d'autres programmes en plus de Perl. En particulier vos programmes externes lancés depuis Perl verront ces changements. Si vous rendez ces nouveaux réglages permanents, tous les programmes que vous ferez tourner les verront. Voir ENVIRONNEMENT (§48.6) pour une liste complète des variables d'environnement pertinentes et UTILISATION DES "LOCALE" pour leurs effets sur Perl. Les effets sur les autres programmes sont aisément déduits. Par exemple, la variables LC\_COLLATE peut très bien influencer votre programme **sort** (ou plutôt le programme qui trie des 'enregistrements' dans l'ordre alphabétique, quel que soit son nom sur votre système).

Vous pouvez tester des changements temporaires sur ces variables puis, lorsque les nouveaux réglages semblent améliorer les choses, placer ces réglages dans l'un des fichiers de démarrage de votre shell. Consultez votre documentation locale pour les détails. Pour les shells de type Bourne-shell (**sh**, **ksh**, **bash**, **zsh**), cela donne :

```
LC_ALL=en_US.ISO8859-1
export LC_ALL
```

Cela suppose que vous avez vu le "locale" "en\_US.ISO8859-1" en utilisant l'une des commandes présentées plus haut. Nous avons décidé d'essayer celui-là à la place du "locale" fautif "En\_US". En shell type C-shell (**csh**, **tcsh**), cela donne :

```
setenv LC_ALL en_US.ISO8859-1
```

Si vous ne connaissez pas le shell que vous utilisez, consultez votre aide en ligne.

### 48.3.6 Résolution permanente des problèmes de "locale"

La méthode plus longue mais définitive de résoudre les problèmes de "locale" est de corriger vous-même la mauvaise installation ou configuration. La correction de la mauvaise (ou inexistante) installation peut nécessiter l'aide d'un administrateur système complaisant.

Tout d'abord, voyez plus haut ce qui concerne Trouver les "locale". Cela vous explique comment savoir les "locale" réellement supportés - et, plus important, installés - sur votre système. Dans notre message d'erreur d'exemple, les variables d'environnement affectant les "locale" sont listées dans l'ordre décroissant d'importance (la variables non positionnées ne comptent pas). Par conséquent, avoir LC\_ALL fixé à "En\_US" doit être un mauvais choix tel que se présente notre message d'erreur. Essayez donc de corriger les réglages de "locale" listés en premier.

En second, si en utilisant les commandes suggérées, vous voyez **exactement** quelque chose comme "En\_US" sans les guillemets alors cela devrait fonctionner puisque vous utilisez alors un "locale" qui est censé être disponible sur votre système. Dans ce cas, voyez Résolution permanente des problèmes de configuration système des "locale".

### 48.3.7 Résolution permanente des problèmes de configuration système des "locale"

C'est lorsque vous voyez quelque chose comme :

```
perl: warning: Please check that your locale settings:
 LC_ALL = "En_US",
 LANG = (unset)
are supported and installed on your system.
```

mais que vous ne trouvez pas "En\_US" dans les listes produites par les commandes mentionnées plus haut. Vous pouvez voir des choses comme "en\_US.ISO8859-1" mais ce n'est pas la même chose. Dans ce cas, essayez le "locale" de la liste qui ressemble le plus à celui que vous aviez précédemment. Les règles pour trouver les noms des "locale" sont un peu vagues car il y a un manque de standardisation dans ce domaine. Voir Trouver les "locale" pour les règles générales.

### 48.3.8 Configuration système des "locale"

Contactez un administrateur système (de préférence le vôtre), rapportez lui exactement le message que vous avez obtenu et demandez lui de lire la documentation que vous lisez présentement. Il devrait être en mesure de trouver ce qui ne va pas dans la configuration des "locale" de votre système. Trouver les "locale" est malheureusement un peu vague à propos des commandes et des emplacements car tout cela n'est pas standardisé.

### 48.3.9 La fonction localeconv

La fonction POSIX::localeconv() vous permet d'obtenir toutes les informations particulières liées au formatage des valeurs numériques selon le "locale" spécifié par LC\_NUMERIC et LC\_MONETARY. (Si vous voulez juste le nom du "locale" courant pour une catégorie particulière, utilisez POSIX::setlocale() avec un seul argument – voir La fonction setlocale.)

```
use POSIX qw(locale_h);

Obtention d'une référence vers une table de hachage
des information dépendant du "locale"
$locale_values = localeconv();

Affichage de la liste triée des valeurs
for (sort keys %$locale_values) {
 printf "%-20s = %s\n", $_, $locale_values->{$_}
}
}
```

localeconv() ne prend aucun argument et retourne **une référence** vers une table de hachage. Les clés de cette table sont des noms des variables de formatage telle que decimal\_point et thousands\_sep. Les valeurs sont celles associées à cette clé. Voir localeconv in POSIX pour un exemple plus long listant les catégories qu'une implémentation devrait fournir; certaines en fournissent plus, d'autres moins. Vous n'avez pas besoin d'un use locale explicite puisque localeconv() respecte toujours le "locale" courant.

Voici un programme d'exemple simple qui réécrit les paramètres de sa ligne de commande comme des entiers correctement formatés selon le "locale" courant :

```
Voir les commentaires des exemples précédents
require 5.004;
use POSIX qw(locale_h);

Obtention de quelques-uns des paramètres
de formatage numérique
my ($thousands_sep, $grouping) =
 @{$localeconv()}{'thousands_sep', 'grouping'};

Valeur par défaut
$thousands_sep = ',' unless $thousands_sep;
```



```

grouping et mon_grouping sont sous la forme d'un liste compacte de
petits entiers (caractères) indiquant le regroupement (thousand_seps
et mon_thousand_seps spécifiant les séparateurs de groupes) des
chiffres dans les nombres et la valeurs monétaires. La signification
de ces entiers est : 255 pour indiquer qu'il ne faut plus regrouper,
0 pour indiquer de répéter le groupe précédent, 1-254 pour indiquer
la taille du groupe courant. Les regroupements ont lieu de la droite
vers la gauche. Dans ce qui suit, nous trichons un peu en
n'utilisant que la première valeur de grouping.
if ($grouping) {
 @grouping = unpack("C*", $grouping);
} else {
 @grouping = (3);
}

Présentation des paramètres de la ligne de commandes
selon le "locale" courant
for (@ARGV) {
 $_ = int; # Suppressions de la partie non entière
 1 while
 s/(\d)(\d{ $grouping[0]}($| $thousands_sep))/ 1thousands_sep$2/;
 print "$_";
}
print "\n";

```

## 48.4 CATÉGORIES DE "LOCALE"

Les sous-sections suivantes décrivent les catégories de base de "locale". En plus, certaines combinaisons de catégories permettent la manipulation de plusieurs catégories à la fois. Voir ENVIRONNEMENT (§48.6).

### 48.4.1 Catégorie LC\_COLLATE: tri

Dans la portée de `use locale`, Perl regarde la variable d'environnement `LC_COLLATE` pour déterminer la notion de collation (ordre) des caractères. Par exemple, 'b' suit 'a' dans l'alphabet Latin mais où se situent 'á' et 'â' ? Et bien que 'color' suive 'chocolate' en Anglais, qu'en est-il en Espagnol ?

Les collations suivantes sont toutes sensées et vous pouvez toutes les rencontrer si vous utilisez "use locale".

```

A B C D E a b c d e
A a B b C c D d D e
a A b B c C d D e E
a b c d e A B C D E

```

Voici un petit bout de code permettant d'afficher les caractères alphanumériques du "locale" courant dans l'ordre de ce "locale" :

```

use locale;
print +(sort grep /\w/, map { chr() } 0..255), "\n";

```

Comparez cela avec les caractères et l'ordre que vous obtenez si vous indiquez explicitement que le "locale" doit être ignoré :

```

no locale;
print +(sort grep /\w/, map { chr() } 0..255), "\n";

```

Cette ordre natif à la machine (qui est celui obtenu à moins qu'apparaisse préalablement de le bloc un `use locale`) doit être utilisé pour trier des données binaires alors que l'ordre dépendant du "locale" du premier exemple est utile pour les textes en langage naturel.

Comme indiqué dans UTILISATION DES "LOCALE", `cmp` effectue sa comparaison selon l'ordre du "locale" courant lorsque `use locale` est actif mais retombe sur une comparaison octet par octet pour les chaînes que le "locale" donne égales. Vous pouvez utiliser `POSIX::strcoll()` si vous ne voulez pas de cette comparaison :

```
use POSIX qw(strcoll);
$equal_in_locale =
 !strcoll("space and case ignored", "SpaceAndCaseIgnored");
```

\$equal\_in\_locale sera vrai si l'ordre du "locale" spécifie un ordre type dictionnaire qui ignore complètement les caractères d'espace et ne tient pas compte de la casse.

Si vous devez comparer l'égalité (au sens du "locale") d'une seule chaîne avec de nombreuses autres chaînes, vous devriez gagner un peu d'efficacité en utilisant POSIX::strxfrm() et eq :

```
use POSIX qw(strxfrm);
$xfrm_string = strxfrm("Mixed-case string");
print "locale collation ignore spaces\n"
 if $xfrm_string eq strxfrm("Mixed-casestring");
print "locale collation ignore hyphens\n"
 if $xfrm_string eq strxfrm("Mixedcase string");
print "locale collation ignore case\n"
 if $xfrm_string eq strxfrm("mixed-case string");
```

strxfrm() prend une chaîne et la transforme pour l'utiliser lors d'une comparaison octet par octet avec d'autres chaînes transformées. Dans les entrailles de Perl, les opérateurs Perl de comparaison dont le comportement est modifié par le "locale" appellent strxfrm() sur chacun de leurs opérands puis effectuent une comparaison octet par octet des chaînes transformées. En appelant strxfrm() explicitement et en utilisant une comparaison ne tenant pas compte du "locale", cet exemple tente de réduire le nombre de transformation. Mais en réalité, cela ne réduit rien : la magie de Perl (voir *Magic Variables in perl*) crée la version transformée d'une chaîne la première fois qu'elle est nécessaire et conserve cette version au cas où elle serait à nouveau nécessaire. Un exemple réécrit tout simplement en utilisant cmp tournera aussi vite. Elle gère aussi correctement les caractères nuls présents dans les chaînes alors que strxfrm() considère le premier caractère nul rencontré comme étant la fin de la chaîne. N'espérez pas que les chaînes transformées soient portables d'un système à un autre – ou même d'une version à une autre d'un même système. En résumé, n'appellez pas strxfrm() directement : laissez Perl le faire pour vous.

Note : use locale n'est pas présent dans plusieurs exemples car il n'est pas nécessaire ; strcoll() et strxfrm() existent uniquement pour générer des résultats dépendant du "locale" par conséquent ils respectent toujours le "locale" courant LC\_COLLATE.

#### 48.4.2 Catégorie LC\_CTYPE: type de caractères

Dans la portée d'un use locale, Perl respecte le réglage du "locale" LC\_CTYPE. Cela contrôle la notion de caractères alphabétiques qu'utilise l'application. Cela affecte la meta-notation Perl \w des expressions rationnelles qui indique un caractère alphanumérique – c'est à dire un caractère alphabétique ou numérique. (Consultez *perlre* pour plus d'information sur les expressions rationnelles.) Grâce à LC\_CTYPE, et selon les réglages de votre "locale", des caractères tels que 'æ', 'ð', 'ß' et 'ø' peuvent être considérés comme des caractères \w.

Le locale LC\_CTYPE fournit aussi une table de conversion de caractères entre majuscules et minuscules. Cela affecte les fonctions de gestion de la casse – lc(), lcfirst(), uc() et ucfirst(), les interpolations dépendant de la casse \l \L, \u et \U dans les chaînes entre guillemets et les substitutions s/// et les motifs d'expressions rationnelles indépendant de la casse grâce au modificateur i.

Finalement, LC\_CTYPE affecte les fonctions de tests POSIX portant sur les classes de caractères – isalpha(), islower() et autres. Par exemple, si vous passez du "locale" "C" vers un "locale" scandinave 7-bit, vous constaterez que "|" passe de la classe ispunct() à la classe isalpha().

**Note** : une définition erronée ou malicieuse de LC\_CTYPE peut amener votre application à considérer comme alphanumérique des caractères qui ne le sont pas clairement. Pour une reconnaissance stricte des lettres et des chiffres – par exemple, dans une commande – les application tenant compte des "locale" devrait utiliser \w à l'intérieur d'un bloc no locale. Voir SÉCURITÉ (§48.5).

#### 48.4.3 Catégorie LC\_NUMERIC: format numérique

Dans la portée d'un use locale, Perl respecte le réglage du "locale" LC\_NUMERIC qui contrôle la manière dont une application doit présenter les nombres pour être lisible par un humain lors de leur affichage par les fonctions printf(), sprintf() et write(). La conversion d'une chaîne vers un nombre par la fonction POSIX::strtod() est aussi affectée. Dans la plupart des implémentations, le seul effet est le changement du caractère utilisé comme séparateur décimal – peut-être

de '.' en ','. Ces fonctions ne tiennent pas compte des raffinements tels que la séparation des milliers et autres. (Voir La fonction localeconv si vous vous intéressez à cela.)

La sortie produite par print() n'est **jamais** affecté par le "locale" courant : son comportement ne dépend pas d'un use locale ou d'un no locale et correspond à celui de printf() avec un "locale" "C". C'est la même chose pour les conversions internes de Perl entre formats de chaînes et formats numériques :

```
use POSIX qw(strtod);
use locale;

$n = 5/2; # Affecte la valeur numérique 2.5 à $n

$a = " $n"; # conversion en chaîne indépendant du "locale"

print "half five is $n\n"; # Affichage indépendant du "locale"

printf "half five is %g\n", $n; # Affichage dépendant du "locale"

print "DECIMAL POINT IS COMMA\n"
 if $n == (strtod("2,5"))[0]; # Conversion dépendante du "locale"
```

#### 48.4.4 Catégorie LC\_MONETARY: format des valeurs monétaires

Le C standard définit la catégorie LC\_MONETARY mais aucune fonction n'est affectée par son contenu. Par conséquent, Perl n'en fera rien. Si vous voulez vraiment utiliser LC\_MONETARY, vous pouvez demander son contenu – voir La fonction localeconv – et utiliser l'information renvoyée pour présenter vos valeurs monétaires. Par contre, vous constaterez sûrement que, aussi volumineuse et complexe que soit cette information, elle ne ne vous suffira pas : la présentation de valeurs monétaires est très difficile.

#### 48.4.5 LC\_TIME

La valeur produite par POSIX::strftime(), qui construit une chaîne contenant une date dans un format lisible par un être humain, est affectée par le "locale" courant LC\_TIME. Donc, avec un "locale" français, la valeur produite par %B (nom complet du mois) pour le premier mois de l'année sera "janvier". Voici comment obtenir la liste des noms longs des mois du "locale" courant :

```
use POSIX qw(strftime);
for (0..11) {
 $long_month_name[$_] =
 strftime("%B", 0, 0, 0, 1, $_, 96);
}
```

Note : use locale n'est pas nécessaire dans cette exemple. En tant que fonction n'existant que pour générer un résultat dépendant du "locale", strftime() respecte toujours le "locale" LC\_TIME.

#### 48.4.6 Autres catégories

La dernière catégorie de "locale" LC\_MESSAGES n'est pas actuellement utilisée par Perl – excepter qu'elle peut affecter le comportement de certaines fonctions de quelques bibliothèques appelées par des extensions ne faisant pas partie de la distribution Perl standard ou du système lui-même ou de ses utilitaires. Remarquez bien que la valeur de \$! et les messages d'erreur produits par des utilitaires externes peuvent être modifiés par LC\_MESSAGES. Si vous voulez des codes d'erreur portable, utilisez %!. Voir *Errno*.

## 48.5 SÉCURITÉ

Bien que la présentation des questions de sécurité en Perl soit faite dans *perlsec*, une présentation de la gestion par Perl des "locale" ne pourrait être complète si elle n'attirait pas votre attention sur les questions de sécurité directement liées aux "locale". Les "locale" – en particulier sur les systèmes qui autorisent les utilisateurs non privilégiés à construire leur propre "locale" – ne sont pas sûrs. Un "locale" malicieux (ou tout simplement mal fait) peut amener une application liée aux "locale" à produire des résultats inattendus. Voici quelques possibilités :

- Les expressions rationnelles vérifiant la validité des noms de fichiers ou des adresses mail en utilisant `\w` peuvent être abusées par un "locale" `LC_CTYPE` qui prétend que des caractères tels que ">" et "|" sont alphanumériques.
- Les interpolations de chaînes avec des conversions majuscules/minuscules telles que `$dest = "C:\U$name.$ext"` peuvent produire de dangereux résultats si une table de conversion `LC_CTYPE` erronée est active.
- Certains systèmes mal conçus autorisent même les utilisateurs à modifier le "locale" "C". Si le séparateur décimal de la catégorie `LC_NUMERIC` du "locale" "C" est sournoisement changé du point vers la virgule alors `sprintf("%g", 0.123456e3)` produira le résultat "123,456". De nombreuses personnes l'interpréteront alors comme cent vingt trois mille quatre cent cinquante six.
- Un "locale" `LC_COLLATE` sournois peut amener les étudiants avec une note "D" à être classés devant ceux ayant un "A".
- Une application se reposant sur les informations de `LC_MONETARY` pourrait produire des débits comme si c'était des crédits et vice versa si son "locale" a été perverti. Ou elle pourrait faire des paiements en dollars US au lieu de dollars de Honk Kong.
- Les dates et le nom des jours dans les dates formatées par `strftime()` pourraient être manipulés avantageusement par un utilisateur malicieux ayant la possibilité de modifier le "locale" `LC_DATE`.

De tels dangers ne sont pas spécifiques au système de "locale" : tous les aspects de l'environnement d'une application qui peuvent être malicieusement modifiés présente les mêmes risques. De même, ils ne sont pas spécifiques à Perl : tous les langages de programmation qui vous permettent d'écrire des programmes qui tiennent compte de leur environnement sont exposés à ces problèmes.

Perl ne peut pas vous protéger de toutes les possibilités présentées dans ces exemples – il n'y a rien de mieux que votre propre vigilance – mais, lorsque `use locale` est actif, Perl utilise le mécanisme de souillure (voir *perlsec*) pour marquer les chaînes qui sont dépendantes du "locale" et qui, par conséquence, ne sont pas sûres. Voici une liste des opérateurs et fonctions dont le comportement vis-à-vis des souillures peut être affecté par le "locale" :

### Opérateurs de comparaison (`lt`, `le`, `ge`, `gt` et `cmp`):

Les scalaires vrai/faux (ou plus petit/égal/plus grand) ne sont jamais souillés.

### Interpolation dépendant de la casse (avec `\l`, `\L`, `\u` ou `\U`)

Les chaînes résultant d'une telle interpolation sont souillées si `use locale` est actif.

### Opérateur de reconnaissance (`m//`):

Les scalaires vrai/faux ne sont jamais souillés.

Les sous-motifs délivrés soit par un résultat dans un contexte de liste soit par `$1`, etc. sont souillés si `use locale` est actif et si le sous-motif de l'expression rationnelle contient `\w` (pour reconnaître un caractère alphanumérique), `\W` (un caractère non alphanumériques), `\s` (un caractère blanc) ou `\S` (un caractère non blanc). Les variables `$&`, `$'`, `$` et `$+` sont aussi souillées si `use locale` est actif et si l'expression rationnelle contient `\w`, `\W`, `\s` ou `\S`.

### Opérateur de substitution (`s///`):

A le même comportement que l'opérateur de reconnaissance. De plus, l'opérande de gauche d'un `=~` devient souillé lorsque `use locale` est actif et si la modification est le résultat d'une substitution basée sur une expression rationnelle impliquant `\w`, `\W`, `\s` ou `\S` ainsi que les changements de casse avec `\l`, `\L`, `\u` ou `\U`.

### Fonctions de présentation (`printf()` et `write()`):

Le résultat (échec ou réussite) n'est jamais souillé.

### Les fonctions de changement de casse (`lc()`, `lcfirst()`, `uc()`, `ucfirst()`):

Le résultat est souillé si `use locale` est actif.

### Les fonction POSIX dépendant du "locale" (`localeconv()`, `strcoll()`, `strftime()`, `strxfrm()`):

Le résultat n'est jamais souillé.

### Les tests de classes de caractères POSIX (`isalnum()`, `isalpha()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`):

Le résultat (vrai/faux) n'est jamais souillé.

Les trois exemples suivant illustrent les souillures dépendant du "locale".

Le premier programme, qui ignore les "locale", ne fonctionne pas : une valeur prise directement de la ligne de commande ne peut être utilisée pour nommer un fichier de sortie lorsque le mode souillé est actif.

```

#/usr/local/bin/perl -T
mode souillé actif (-T)

Omission de la vérification de la ligne de commande
$tainted_output_file = shift;

open(F, ">$tainted_output_file")
 or warn "Open of $untainted_output_file failed: $!\n";

```

Ce programme peut être transformé afin de "blanchir" la valeur souillée grâce à une expression rationnelle : le deuxième exemple – qui ignore encore les informations du "locale" – fonctionne et crée le fichier dont le nom est donné sur la ligne de commande si il le peut.

```

#/usr/local/bin/perl -T

$tainted_output_file = shift;
$tainted_output_file =~ m%[\w/]+%;
$untainted_output_file = $&;

open(F, ">$untainted_output_file")
 or warn "Open of $untainted_output_file failed: $!\n";

```

Comparez avec le programme suivant qui, lui, prend en compte les "locale" :

```

#/usr/local/bin/perl -T

$tainted_output_file = shift;
use locale;
$tainted_output_file =~ m%[\w/]+%;
$localized_output_file = $&;

open(F, ">$localized_output_file")
 or warn "Open of $localized_output_file failed: $!\n";

```

Ce troisième programme ne peut pas fonctionner parce que \$& est souillée : c'est le résultat d'une reconnaissance impliquant \w alors que use locale est actif.

## 48.6 ENVIRONNEMENT

### PERL\_BADLANG

Une chaîne qui peut supprimer l'avertissement de Perl au démarrage au sujet des réglages de "locale" incorrectes. L'échec peut provenir d'un support défectueux des "locale" sur votre système – ou d'une faute de frappe dans le nom du "locale" lors de l'écriture de votre environnement. Si cette variable d'environnement est absente ou si sa valeur ne vaut pas zéro – "0" ou "" – Perl se plaindra au sujet des réglages de "locale" incorrectes.

**Note** : PERL\_BADLANG ne vous donne qu'un moyen de cacher le message d'avertissement. Ce message vous signale un problème de votre système de "locale" et vous devriez pousser vos investigations pour savoir d'où vient ce problème.

Les variables d'environnement suivantes ne sont pas spécifiques à Perl : elles font parties de la méthode standard (ISO C, XPG4, POSIX 1.c) setlocale() permettant de contrôler la manière dont les applications gèrent les données.

### LC\_ALL

LC\_ALL est la variable d'environnement "cache-tout-le-reste". Si elle existe, elle cache toutes les autres variables d'environnement liées au "locale".

### LANGUAGE

**NOTE** : LANGUAGE est une extension GNU. Elle ne vous affectera que si vous utilisez la libc GNU. C'est le cas si vous utilisez Linux. Si vous utilisez un UNIX "commercial", vous n'utilisez probablement pas la libc GNU et vous pouvez ignorer LANGUAGE.

En revanche, si votre bibliothèque utilise LANGUAGE, cela affectera la langue des messages d'information, d'avertissement et d'erreur produits par vos commandes (autrement dit, c'est comme LC\_MESSAGES) en ayant une priorité plus élevée que LC\_ALL. De plus, ce n'est une simple valeur mais une suite (une liste séparée par ":") de langues (pas des "locale"). Voir gettext dans la documentation de votre bibliothèque GNU pour plus d'information.

**LC\_CTYPE**

En l'absence de LC\_ALL, LC\_CTYPE détermine le type de caractères du "locale". En l'absence de LC\_ALL et de LC\_CTYPE, LANG est utilisé pour choisir le type de caractères.

**LC\_COLLATE**

En l'absence de LC\_ALL, LC\_COLLATE détermine l'ordre (de tri) du "locale". En l'absence de LC\_ALL et de LC\_COLLATE, LANG est utilisé pour choisir l'ordre.

**LC\_MONETARY**

En l'absence de LC\_ALL, LC\_MONETARY détermine les réglages monétaires du "locale". En l'absence de LC\_ALL et de LC\_MONETARY, LANG est utilisé pour choisir les réglages monétaires.

**LC\_NUMERIC**

En l'absence de LC\_ALL, LC\_NUMERIC détermine l'affichage des nombres du "locale". En l'absence de LC\_ALL et de LC\_NUMERIC, LANG est utilisé pour choisir l'affichage des nombres.

**LC\_TIME**

En l'absence de LC\_ALL, LC\_TIME détermine l'affichage des dates et heures du "locale". En l'absence de LC\_ALL et de LC\_TIME, LANG est utilisé pour choisir l'affichage des dates et heures.

**LANG**

LANG est la variable d'environnement générale des "locale". Si elle est positionnée, elle est utilisée en dernier recours après LC\_ALL et LC\_... spécifique à chaque catégorie.

## 48.7 NOTES

### 48.7.1 Compatibilité

Les versions de Perl antérieures à 5.004 ignorent la plupart du temps les informations de "locale" et adoptent un comportement général similaire au choix du "locale" "C" même si l'environnement du programme suggère autre chose (voir `La` fonction `setlocale`). Par défaut, Perl conserve ce comportement pour des raisons de compatibilité. Si vous voulez qu'une application Perl tienne compte aux informations de "locale", vous **devez** utiliser la directive `use locale` pour lui dire.

Les versions de Perl 5.002 et 5.003 utilisaient l'information LC\_CTYPE si elle était présente. C'est à dire que `\w` comprenait ce qui était une lettre au sens du "locale" indiqué par les variables d'environnement. Mais l'utilisateur n'avait aucun contrôle sur cette fonctionnalité : si la bibliothèque C supportait les "locale", Perl les utilisait.

### 48.7.2 I18N:Collate obsolète

Dans les versions de Perl antérieures à 5.004, il était possible de gérer des ordres liés aux "locale" grâce au module `I18N::Collate`. Ce module est maintenant devenu largement obsolète et devrait être évité dans de nouvelles applications. La fonctionnalité LC\_COLLATE est maintenant complètement intégrée dans le coeur du langage Perl.

### 48.7.3 Impactes sur la vitesses des tris et sur l'utilisation de la mémoire

Comparer et trier en respectant le "locale" est normalement plus lent que le tri par défaut. Des temps deux à quatre fois plus longs ont été observés. Ce consomme aussi plus de mémoire : une fois qu'une variable scalaire de Perl a participé à une comparaison de chaîne ou à une opération de tri respectant les règles de tri du "locale", elle peut occuper de 3 à 15 fois plus de mémoire qu'avant (le facteur multiplicatif exact dépend du contenu de la chaîne, du système d'exploitation et du "locale"). Cette dégradation est plus dictée par l'implémentation des "locale" dans le système d'exploitation que par Perl.

### 48.7.4 `write()` et LC\_NUMERIC

Les formats constituent le seul endroit où Perl utilise inconditionnellement les informations du "locale". Si l'environnement du programme spécifie un "locale" LC\_NUMERIC, il est toujours utilisé pour spécifier le séparateur décimale des données sorties par format. Les sorties formatées ne peuvent pas être contrôlées par `use locale` car cette directive est rattachée à la structure de bloc du programme alors que, pour des raisons historiques, les formats existent en dehors de cette structure.

### 48.7.5 Définitions de "locale" disponibles librement

Vous trouverez une grosse collection de définitions de "locale" sur <ftp://dkuug.dk/i18n/WG15-collection>. Il n'y a aucune garantie de fonctionnement ni aucun support. Si votre système d'exploitation accepte l'installation de "locale" arbitraire, cela peut vous être utile soit comme "locale" directement utilisables soit comme base de développement de vos propres "locale".

### 48.7.6 I18n et I10n

"Internationalization" est souvent abrégé en **i18n** puisque la première et la dernière lettres sont séparées par 18 autres lettres. De la même manière, "localization" donne **I10n**.

### 48.7.7 Un standard imparfait

L'internationalisation, telle que définie par les standard C et POSIX, peut être considérée comme incomplète, de faible gain et d'une granularité trop grossière (les "locale" s'appliquent à l'ensemble d'un process alors qu'ils devraient pouvoir s'appliquer à un seul fil (thread), à un seul groupe de fenêtres ou autres). Elle a aussi tendance, comme dans tous les groupes de standardisation, à diviser le monde en nations alors que nous savons tous qu'il peut aussi se diviser en catégories telles les banquiers, les cyclistes, les joueurs, etc. Mais, pour l'instant, c'est le seul standard que nous ayons. Cela peut être considéré comme un bug.

## 48.8 BUGS

### 48.8.1 Les systèmes bugués

Sur certains systèmes, le support des "locale" est bugué et ne peut être corrigé ou utilisé par Perl. De telles déficiences peuvent aboutir à de mystérieux plantage de Perl lorsque `locale` est actif. Si vous êtes confronté à un tel système, rappez ces détails insupportables à [<perlbug@perl.com>](mailto:perlbug@perl.com) et plaignez-vous auprès de votre vendeur : peut-être existe-t-il des correctifs pour votre système. Parfois, ces correctifs s'appellent une mise à jour.

## 48.9 VOIR AUSSI

isalnum in *POSIX*  
 isalpha in *POSIX*  
 isdigit in *POSIX*  
 isgraph in *POSIX*  
 islower in *POSIX*  
 isprint in *POSIX*,  
 ispunct in *POSIX*  
 isspace in *POSIX*  
 isupper in *POSIX*,  
 isxdigit in *POSIX*  
 localeconv in *POSIX*  
 setlocale in *POSIX*,  
 strcoll in *POSIX*  
 strftime in *POSIX*  
 strtod in *POSIX*,  
 strxfrm in *POSIX*

## 48.10 HISTORIQUE

Document original *perli18n.pod* de Jarkko Hietaniemi modifié par Dominic Dunlop assisté des perl5-porters. Prose un peu améliorée par Tom Christiansen.

Dernière mise à jour : Thu Jun 11 08:44:13 MDT 1998

## **48.11 TRADUCTION**

### **48.11.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **48.11.2 Traducteur**

Paul Gaborit <Paul.Gaborit @ enstimac.fr>

### **48.11.3 Relecture**

Personne pour l'instant.



# Chapitre 49

## perluniintro

Introduction à l'utilisation d'Unicode en Perl

### 49.1 DESCRIPTION

Ce document donne une idée générale d'Unicode et de son utilisation en Perl.

#### 49.1.1 Unicode

Unicode est un jeu de caractères standardisé qui prévoit de codifier tous les systèmes d'écriture existant de par le monde ainsi que de nombreux autres symboles.

Unicode et ISO/IEC 10646 sont des normes qui attribuent un code pour les caractères de tous les jeux de caractères modernes, couvrant plus de 30 systèmes d'écriture et des centaines de langues, incluant toutes les langues modernes commercialement importantes. Tous les caractères chinois, japonais et coréens en font aussi parti. Ces standards prévoient de couvrir tous les caractères de plus de 250 systèmes d'écriture et de milliers de langues. Unicode 1.0 est sorti en octobre 1991 et sa version 4.0 en avril 2003.

Un *caractère* Unicode est une entité abstraite. Il n'est lié à aucune taille d'entier en particulier et encore moins au type `char` du langage C. Unicode est neutre vis à vis de la langue et de l'affichage : il ne code pas la langue utilisée par le texte et ne définit pas de polices ou d'autres aspects de présentation graphique. Unicode concerne les caractères et le texte composé de ces caractères.

Unicode définit des caractères comme `LATIN CAPITAL LETTER A` ou `GREEK SMALL LETTER ALPHA` et un nombre unique pour chacun de ces caractères, en l'occurrence respectivement `0x0041` et `0x03B1`. Ces nombres uniques sont appelés *points de code* ou plus simplement *numéros de caractères*.

Le standard Unicode préfère la notation hexadécimale pour les numéros de caractères. Si des nombres tels que `0x0041` ne vous sont pas familiers, jetez un oeil à la section [Notation hexadécimale \(§49.1.14\)](#). Le standard Unicode utilise la notation `U+0041 LATIN CAPITAL LETTER A` pour donner le numéro hexadécimal du caractère suivi de son nom normalisé.

Unicode définit aussi différentes *propriétés* pour les caractères, telles que "uppercase" (majuscule) ou "lowercase" (minuscule), "decimal digit" (chiffre décimal) ou "punctuation" (ponctuation); ces propriétés étant indépendantes du nom des caractères. En outre, sont définies plusieurs opérations sur les caractères telles que "passer en majuscule", "passer en minuscule" et "trier".

Un caractère Unicode est constitué soit d'un unique caractère, soit d'un *caractère de base* (tel que `LATIN CAPITAL LETTER A`) suivi d'un ou plusieurs *modificateurs* ou *caractères combinatoires* (tel que `COMBINING ACUTE ACCENT`). Cette séquence composée d'un caractère de base suivi de ses modificateurs est appelée une *séquence combinante*.

Le fait de considérer ces séquences comme étant un "caractère" dépend de votre point de vue. En tant que programmeur, vous aurez plutôt tendance à considérer chaque élément de la séquence pris séparément. Alors qu'un utilisateur considérera probablement la séquence complète comme étant un unique "caractère" car c'est de cette manière qu'il est perçu dans sa langue.

Avec la vision globale, déterminer le nombre total de caractères est difficile. Mais avec le point de vue du programmeur (chaque caractère, qu'il soit de base ou combinatoire, est un caractère), le concept de nombre de caractères est plus déterministe. Dans ce document, nous adopterons ce second point de vue : un "caractère" est un point de code Unicode, que ce soit un caractère de base ou un caractère combinatoire.

Pour certaines combinaisons, il existe des caractères *précomposés*. Par exemple LATIN CAPITAL LETTER A WITH ACUTE est défini comme un point de code unique. Ces caractères précomposés ne sont disponibles que pour certaines combinaisons et le sont principalement pour faciliter la conversion entre Unicode et les anciens standards (tel que ISO 8859). De manière générale, la méthode par combinaison est plus extensible. Pour faciliter la conversion entre les différentes manières de composer un caractère, plusieurs *formes de normalisation* sont définies dans le but de standardiser les représentations.

Pour des raisons de compatibilité avec les anciens systèmes de codage, l'idée "un numéro unique pour chaque caractère" est un peu remise en cause : en fait, il y a "au moins un numéro pour chaque caractère". Un même caractère peut être représenté différemment dans plusieurs anciens systèmes d'encodage. De même certains points de code ne correspondent à aucun caractère. En effet, d'une part, des blocs déjà utilisés contiennent des points de code non alloués. Et d'autre part, dans Unicode, on trouve des caractères de contrôle (ou de commande) et des caractères spéciaux qui ne représentent pas de vrais caractères.

Un mythe courant concernant Unicode est que celui-ci serait "16 bits" car il n'existe que `0x10000` (ou 65536) caractères entre `0x0000` et `0xFFFF`. **Ceci est faux.** Depuis Unicode 2.0 (juillet 1996), Unicode utilise 21 bits (`0x10FFFF`) et depuis Unicode 3.1 (mars 2001), des caractères ont été définis au-delà de `0xFFFF`. Les `0x10000` premiers caractères sont appelés *plan 0*, ou encore *plan multilingue de base* (PMP – ou BMP en anglais). Avec Unicode 3.1, c'est 17 (oui, dix-sept) plans qui ont été définis mais ils ne sont pas tous entièrement remplis, du moins pour le moment.

Un autre mythe est qu'il existerait une corrélation entre les langues et les blocs de 256 caractères, chaque bloc définissant les caractères utilisés par une langue ou un ensemble de langues. **Ceci est tout aussi faux.** Cette division en bloc existe, mais elle est presque totalement accidentelle, un simple artéfact de la manière dont les caractères ont été et continuent à être alloués. En revanche, il existe le concept d'*écriture* (ou script en anglais) qui est plus utile : il existe une écriture latine, une grecque et ainsi de suite. Les écritures sont généralement réparties sur plusieurs blocs. Pour plus d'information consultez *Unicode::UCD*.

Les point de code Unicode ne sont que des nombres abstraits. Pour recevoir ou émettre ces nombres abstraits, ils faut les *encoder* ou les *sérialiser* d'une manière ou d'une autre. Unicode définit plusieurs *formes d'encodage*, parmi lesquelles UTF-8 est peut-être la plus populaire. UTF-8 est méthode d'encodage à taille variable qui encode les caractères Unicode sur 1 à 6 octets (maximum 4 pour les caractères actuellement définis). Les autres méthodes incluent UTF-16, UTF-32 et leur variantes grand-boutistes et petit-boutistes (UTF-8 est indépendant de l'ordre des octets). L'ISO/IEC définit aussi les formes UCS-2 et UCS-4.

Pour plus d'information à propos de l'encodage, par exemple pour savoir à quoi correspondent les caractères de substitution (*surrogates*) et les *marques d'ordre d'octets* (*byte order marks* ou BOMs), consultez *perlunicode*.

### 49.1.2 Le support d'Unicode en Perl

À partir de la version 5.6.0, Perl était apte à gérer Unicode nativement. Mais la version 5.8.0 a été la première version recommandée pour pouvoir travailler sérieusement avec Unicode. La version de maintenance 5.6.1 a corrigé un grand nombre des problèmes de l'implémentation initiale, mais par exemple, les expressions rationnelles ne fonctionnaient toujours pas en Unicode dans la version 5.6.1.

**À partir de Perl 5.8.0, l'utilisation de `use utf8` n'est plus nécessaire.** Dans les versions précédentes le pragma `utf8` était utilisé pour déclarer que les opérations du bloc ou du fichier courant étaient compatibles Unicode. Ce modèle s'avéra mauvais, ou tout du moins maladroit : la caractéristique "Unicode" est maintenant liée aux données et non plus aux opérations. L'utilisation de `use utf8` ne reste nécessaire que dans un seul cas : lorsque le script Perl est lui-même encodé en UTF-8. Cela vous permet alors d'utiliser l'UTF-8 pour vos identifiants ainsi que dans les chaînes de caractères et les expressions rationnelles. Ceci n'est pas le réglage par défaut, car sinon les scripts comportant des données encodées en 8 bits risqueraient de ne plus fonctionner. Consultez *utf8*.

### 49.1.3 Le modèle Unicode de Perl

Perl supporte aussi bien les chaînes de caractères utilisant l'encodage pré-5.6 sur huit bits natifs que celles utilisant des caractères Unicode. Le principe est que Perl essaye au maximum de conserver les données avec l'encodage 8 bits, mais dès que Unicode devient indispensable, les données sont converties en Unicode de manière transparente.

En interne, Perl utilise le jeu de caractères natif sur 8 bit de la plateforme (par exemple Latin-1) et pour encoder les chaînes de caractères Unicode, il utilise par défaut UTF-8. Plus précisément, si tous les points de codes d'une chaîne sont inférieurs ou égal à `0xFF`, Perl utilisera l'encodage sur 8 bits natif. Sinon, il utilisera UTF-8.

En général, l'utilisateur de Perl n'a besoin ni de savoir ni de se préoccuper de la manière dont Perl encode ses chaînes de caractères en interne, mais cela peut devenir utile lorsqu'on envoie des chaînes Unicode dans un flux n'utilisant pas PerlIO (il utilisera donc l'encodage par "défaut"). Dans ce cas, les octets bruts utilisés en interne (reposant sur le jeu de

caractères natif ou sur UTF-8, selon les chaînes) seront transmis et un avertissement "Wide character" sera émis pour les chaînes contenant un caractère dépassant 0x00FF.

Par exemple,

```
perl -e 'print "\x{DF}\n", "\x{0100}\x{DF}\n"'
```

produit un mélange plutôt inutile de caractères natifs et d'UTF-8 ainsi que l'avertissement :

```
Wide character in print at ...
```

Pour afficher de l'UTF-8, il faut utiliser la couche `:utf8`. L'ajout de :

```
binmode(STDOUT, ":utf8");
```

au début de ce programme d'exemple forcera l'affichage à être en totalité en UTF-8 et retirera les avertissements.

Vous pouvez automatiser l'utilisation de l'UTF-8 pour vos entrées/sorties standard, pour les appels à `open()` et pour `@ARGV` en utilisant l'option `-C` de la ligne de commande ou la variable d'environnement `PERL_UNICODE`. Consultez *perlrun* pour la documentation de l'option `-C`.

Notez que cela signifie que Perl espère que les autres logiciels fonctionnent eux-aussi en UTF-8 : si Perl est amené à croire que `STDIN` est en UTF-8 alors que `STDIN` provient d'une commande qui n'est pas en UTF-8, Perl se plaindra de caractères UTF-8 mal composés.

Toutes les fonctionnalités qui combinent Unicode et les E/S (I/O) nécessitent l'usage de la nouvelle fonctionnalité `PerlIO`. La quasi totalité des plateformes Perl 5.8 utilise `PerlIO` mais vous pouvez le vérifier en lançant la commande "perl -V" et en recherchant `useperlio=define`.

#### 49.1.4 Unicode et EBCDIC

Perl 5.8.0 supporte aussi Unicode sur les plateformes EBCDIC. Le support d'Unicode y est plus complexe à implémenter car il nécessite des conversions supplémentaires à chaque utilisation. Certains problèmes persistent ; consultez *perlebcdic* pour plus de détails.

Dans tous les cas, le support actuel d'Unicode sur les plateformes EBCDIC est bien meilleur que dans les versions 5.6 (cela ne fonctionnait quasiment pas sur les plateformes EBCDIC). Sur ces plateformes, l'encodage interne est UTF-EBCDIC au lieu d'UTF-8. La différence est que UTF-8 est "ASCII-sûr" dans le sens où les caractères ASCII sont codés tels quels en UTF-8, alors que UTF-EBCDIC est "EBCDIC-sûr".

#### 49.1.5 Créer de l'Unicode

Dans les littéraux, pour créer des caractères Unicode dont le point de code est supérieur à 0xFF, il faut utiliser la notation `\x{...}` dans une chaîne entre guillemet :

```
my $smiley = "\x{263a}";
```

De la même façon, cette notation peut être utilisée dans les expressions rationnelles :

```
$smiley =~ /\x{263a}/;
```

À l'exécution vous pouvez utiliser `chr()` :

```
my $hebrew_alef = chr(0x05d0);
```

Consultez Autres ressources (§49.1.15) pour savoir comment trouver ces codes numériques.

Évidemment, `ord()` fera l'inverse : il transformera un caractère en un point de code.

Notez que `\x..` (sans `{}` et avec seulement deux chiffres hexadécimaux), `\x{...}` ainsi que `chr(...)` génèrent un caractère sur un octet pour les arguments inférieurs à 0x100 (256 en décimal) et ceci pour conserver une compatibilité ascendante avec les anciennes versions de Perl. Pour les arguments supérieurs à 0xFF, il s'agira toujours de caractères Unicode. Si vous voulez forcer la génération de caractères Unicode quelle que soit la valeur, utilisez `pack("U", ...)` au lieu de `\x..`, `\x{...}`, ou `chr()`.

Vous pouvez aussi utiliser le pragma `chardnames` pour invoquer les caractères par leur nom entre guillemets :

```
use charnames ':full';
my $arabic_alef = "\N{ARABIC LETTER ALEF}";
```

Et, comme indiqué précédemment, vous pouvez aussi utiliser la fonction `pack()` pour produire des caractères Unicode :

```
my $georgian_an = pack("U", 0x10a0);
```

Notez que `\x{...}` et `\N{...}` sont des constantes : vous ne pouvez pas utiliser de variables dans ces notations. Si vous voulez un équivalent dynamique, utilisez `chr()` et `charnames::vianame()`.

Si vous voulez forcer un résultat en caractères Unicode, utilisez le préfixe spécial "U0". Il ne consomme pas d'argument mais force, dans le résultat, l'utilisation de caractères Unicode à la place d'octets.

```
my $chars = pack("U0C*", 0x80, 0x42);
```

De la même manière, vous pouvez forcer l'utilisation d'octets par le préfixe spécial "C0".

### 49.1.6 Manipuler l'Unicode

La manipulation d'Unicode est quasi transparente : il suffit d'utiliser les chaînes de caractères comme d'habitude. Les fonctions comme `index()`, `length()`, et `substr()` fonctionneront avec des caractères Unicode et il en sera de même avec les expressions rationnelles (consultez *perlunicode* et *perlretut*).

Notez que Perl considère les séquences combinantes comme étant composées de caractères séparés, par exemple :

```
use charnames ':full';
print length("\N{LATIN CAPITAL LETTER A}\N{COMBINING ACUTE ACCENT}"), "\n";
```

affichera 2 et non 1. La seule exception est l'utilisation de `\X` dans les expressions rationnelles pour reconnaître une séquence combinante de caractères.

En revanche, les choses ne sont malheureusement pas toujours aussi transparentes que ce soit avec l'encodage natif, avec les E/S ou dans certaines situations spéciales. C'est que nous allons détaillé maintenant.

### 49.1.7 Encodages natifs

Lorsque vous utilisez conjointement des données natives et des données Unicode, les données natives doivent être converties en Unicode. Par défaut ISO 8859-1 (ou EBCDIC selon le cas) est utilisé. Vous pouvez modifier ce comportement en utilisant le pragma `encoding`, par exemple :

```
use encoding 'latin2'; # ISO 8859-2
```

Dans ce cas, les littéraux (chaînes et expressions rationnelles), `chr()` et `ord()` produiront de l'Unicode en supposant un codage initiale en ISO 8859-2. Notez que le nom de l'encodage est reconnu de manière indulgente : au lieu de `latin2` on aurait pu utiliser `Latin2`, ou `iso8859-2` ou encore d'autres variantes. En utilisant seulement :

```
use encoding;
```

la valeur de la variable `PERL_ENCODING` sera alors utilisée. Si cette variable n'est pas positionnée, le pragma échouera.

Le module `Encode` connaît de nombreux encodages et fournit une interface pour faire des conversions entre eux :

```
use Encode 'decode';
$data = decode("iso-8859-3", $data); # convertit du natif vers utf-8
```

### 49.1.8 Entrées/Sorties et Unicode

Normalement, l'affichage de données Unicode

```
print FH $sune_chaine_avec_unicode, "\n";
```

produit les octets bruts utilisés en interne par Perl pour encoder la chaîne Unicode. Le système d'encodage interne dépend du système ainsi que des caractères présents dans la chaîne. S'il y a au moins un caractère ayant un code supérieur ou égal à 0x100, vous obtiendrez un avertissement. Pour être sûr que la sortie utilise explicitement l'encodage que vous désirez, et pour éviter l'avertissement, ouvrez le flux avec l'encodage désiré. Quelques exemples :

```
open FH, ">:utf8", "fichier";

open FH, ">:encoding(ucs2)", "fichier";
open FH, ">:encoding(UTF-8)", "fichier";
open FH, ">:encoding(shift_jis)", "fichier";
```

et sur les flux déjà ouverts, utilisez `binmode()` :

```
binmode(STDOUT, ":utf8");

binmode(STDOUT, ":encoding(ucs2)");
binmode(STDOUT, ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(shift_jis)");
```

Les noms d'encodages sont peu regardants : la casse n'a pas d'importance et beaucoup possèdent plusieurs alias. Notez par contre que le filtre `:utf8` doit toujours être spécifiée exactement de cette manière ; il n'est *pas* reconnue de manière aussi permissive que les noms d'encodages.

Consultez *PerlIO* pour en savoir plus sur le filtre `:utf8`, *PerlIO::encoding* et *Encode::PerlIO* pour tout ce qui concerne le filtre `:encoding()` et *Encode::Supported* pour les nombreux encodages reconnus par le module *Encode*.

La lecture d'un fichier que vous savez être encodés en Unicode ou en natif ne convertira pas magiquement les données en Unicode aux yeux de Perl. Pour cela, spécifier la couche désirée à l'ouverture du fichier :

```
open(my $fh, '<:utf8', 'anything');
my $ligne_unicode = <$fh>;

open(my $fh, '<:encoding(Big5)', 'anything');
my $ligne_unicode = <$fh>;
```

Pour plus de flexibilité, le filtre E/S peut aussi être spécifié via le pragma `open`. Consultez *open* ou l'exemple suivant :

```
use open ':utf8'; # les entrees et sorties seront par default en UTF-8
open X, ">file";
print X chr(0x100), "\n";
close X;
open Y, "<file";
printf "%#x\n", ord(<Y>); # Cela devrait afficher 0x100
close Y;
```

Avec le pragma `open` vous pouvez utiliser le filtre `:locale` :

```
BEGIN { $ENV{LC_ALL} = $ENV{LANG} = 'ru_RU.KOI8-R' }
:locale va utiliser les variables d'environnements des locales comme LC_ALL
use open OUT => ':locale'; # russki parusski
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # ceci devrais afficher 0xc1
close I;
```

ou vous pouvez aussi utiliser le filtre `:encoding(...)` :

```
open(my $epic, '<:encoding(iso-8859-7)', 'iliad.greek');
my $ligne_unicode = <$epic>;
```

Ces méthodes installent, sur le flux d'entrée/sortie, un filtre transparent qui, lors de leur lecture à partir du flux, convertit les données depuis l'encodage spécifié. Le résultat est toujours de l'Unicode.

En instaurant un filtre par défaut, le pragma `open` impacte tous les appels à `open()` qui suivront. Si vous ne voulez modifier que certains flux, utilisez les filtres explicites directement dans les appels à `open()`.

Vous pouvez modifier l'encodage d'un flux déjà ouvert en utilisant `binmode()` ; consultez `binmode` in *perlfunc*.

Le filtre `:locale` ne peut pas, à l'heure actuelle (comme dans Perl 5.8.0), fonctionner avec `open()` et `binmode()` mais seulement avec le pragma `open`. Par contre `:utf8` et `:encoding(...)` fonctionnent avec `open()`, `binmode()` et le pragma `open`.

De la même manière, vous pouvez utiliser les filtres d'E/S sur un flux en sortie pour convertir automatiquement depuis Unicode vers l'encodage spécifié, lors des écritures dans ce flux. Par exemple, le code suivant copie le contenu du fichier "text.jis" (encodé en ISO-2022-JP, autrement dit JIS) dans le fichier "text.utf8", encodé en UTF-8 :

```
open(my $nihongo, '<:encoding(iso-2022-jp)', 'text.jis');
open(my $unicode, '>:utf8', 'text.utf8');
while (<$nihongo>) { print $unicode $_ }
```

Les noms des encodages fournis à `open()` ou au pragma `open`, sont similaires à ceux du pragma `encoding` en ce qui concerne la flexibilité : `koi8-r` et `KOI8R` seront interprétés de la même manière.

Les encodages communément reconnus par l'ISO, en MIME ou par l'IANA et divers organismes de normalisation sont aussi Sreconnus ;> pour une liste plus détaillée consultez *Encode::Supported*.

La fonction `read()` lit des caractères et renvoie le nombre de caractère lus. Les fonctions `seek()` et `tell()` utilisent un nombre d'octets ainsi que `sysread()` et `sysseek()`.

Notez qu'à cause du comportement par défaut qui consiste à ne pas faire de conversion si aucun filtre par défaut n'est défini, il est facile d'écrire du code qui fera grossir un fichier en ré-encodant les données déjà encodées :

```
ATTENTION CODE DEFECTUEUX
open F, "file";
local $/; ## lis l'ensemble du fichier sous forme de caractères 8 bits
$t = <F>;
close F;
open F, ">:utf8", "file";
print F $t; ## conversion en UTF-8 sur la sortie
close F;
```

Si vous exécutez deux fois ce code, le contenu de *file* sera doublement encodé en UTF-8. L'utilisation de `use open ':utf8'` aurait permis d'éviter ce bug ou on aurait pu aussi ouvrir *file* explicitement en UTF-8.

**NOTE:** Les fonctionnalités `:utf8` et `:encoding` ne fonctionnent que si votre Perl a été compilé avec la nouvelle fonctionnalité PerlIO (ce qui est le cas par défaut sur la plupart des systèmes).

### 49.1.9 Afficher de l'Unicode sous forme de texte

Il peut arriver que vous vouliez afficher des scalaires Perl contenant de l'Unicode comme un simple texte ASCII (ou EBCDIC). La sous-routine suivante convertit ses arguments de telle manière que les caractères Unicode dont le point de code est supérieur à 255 sont affichés sous la forme `\x{...}`, que les caractères de contrôle (comme `\n`) sont affichés sous la forme `\x..` et que le reste est affiché sans conversion :

```
sub nice_string {
 join("",
 map { $_ > 255 ? # Si caractère étendu ...
 sprintf("\x{%04X}", $_) : # \x{...}
 chr($_) =~ /[[:cntrl:]]/ ? # Si caractère de contrôle ...
 sprintf("\x%02X", $_) : # \x..
 quotemeta(chr($_)) # Sinon sous forme quoted
 } unpack("U*", $_[0]));
}
```

Par exemple,

```
nice_string("foo\x{100}bar\n")
```

renvoie la chaîne :

```
'foo\x{0100}bar\x0A'
```

qui est prête à être imprimée.

### 49.1.10 Cas spéciaux

- Opérateur complément bit à bit `~` et `vec()`

L'opérateur de complément bit à bit `~` peut produire des résultats surprenants s'il est utilisé sur une chaîne contenant des caractères dont le point de code est supérieur à 255. Dans un tel cas, le résultat sera conforme à l'encodage interne, mais pas avec grand chose d'autre. Alors, ne le faites pas. Le même problème se pose avec la fonction `vec()` : elle agit sur le motif de bits utilisé en interne pour stocker les caractères Unicode et non sur la valeur du point de code, ce qui n'est probablement pas ce que vous souhaitez.

- Découvrir l'encodage interne de Perl

Les utilisateurs normaux de Perl ne devraient jamais se préoccuper de la manière dont celui-ci encode les chaînes Unicode (parce que la bonne manière de récupérer le contenu d'une chaîne Unicode, en entrée et en sortie, devrait toujours être via un filtre d'E/S explicitement indiqué). Mais si besoin est, voici deux manières de regarder derrière le rideau.

Une première manière d'observer l'encodage interne des caractères Unicode est d'utiliser `unpack("C*", ...)` pour récupérer les octets ou `unpack("H*", ...)` pour les afficher :

```
Affiche c4 80 pour le code UTF-8 0xc4 0x80
print join(" ", unpack("H*", pack("U", 0x100))), "\n";
```

Une autre manière de faire est d'utiliser le module `Devel::Peek` :

```
perl -MDevel::Peek -e 'Dump(chr(0x100))'
```

qui affiche, dans `FLAGS`, le drapeau `UTF8` et, dans `PV`, les deux octets UTF-8 ainsi que le caractère Unicode. Consultez aussi la section concernant la fonction `utf8::is_utf8()` plus loin dans ce document.

### 49.1.11 Sujets avancés

- Équivalence des chaînes

La question de l'équivalence ou de l'égalité de deux chaînes se complique singulièrement avec Unicode : que voulez-vous dire par "égale" ?

(LATIN CAPITAL LETTER A WITH ACUTE est-il égal à LATIN CAPITAL LETTER A ?)

La réponse courte est que par défaut pour les équivalence (`eq`, `ne`) Perl ne se base que sur les points de codes des caractères. Donc, dans le cas ci-dessus, la réponse est non (car `0x00C1 != 0x0041`). Mais quelque fois, toutes les CAPITAL LETTER devrait être considérés comme égale, ou même tous les A.

Pour une réponse détaillée, il faut considérer la normalisation et les problèmes de casse : consultez *Unicode::Normalize*, Unicode Technical Reports #15 and #21, *Unicode Normalization Forms* et *Case Mappings*, <http://www.unicode.org/unicode/reports/tr15/> et <http://www.unicode.org/unicode/reports/tr21/>.

Depuis la version 5.8.0. de Perl, le cas "Full" case-folding de *Case Mappings/SpecialCasing* est implémenté.

- Ordre ou tri des chaînes

Le gens aiment que leurs chaînes de caractères soient correctement triées, le terme anglais utilisé pour Unicode est "collated". Mais, que veux dire trier ?

(Doit-on placé LATIN CAPITAL LETTER A WITH ACUTE avant ou après LATIN CAPITAL LETTER A WITH GRAVE ?)

La réponse courte est que par défaut, les opérateurs de comparaison de chaîne (`lt`, `le`, `cmp`, `ge`, `gt`) de Perl ne se basent que sur le point de code des caractères. Dans le cas précédent, la réponse est "après", car `0x00C1 > 0x00C0`.

La réponse détaillée est "ça dépend", et une bonne réponse ne peut être donnée sans connaître (au moins) le contexte linguistique. Consultez *Unicode::Collate*, et *Unicode Collation Algorithm* <http://www.unicode.org/unicode/reports/tr10/>

### 49.1.12 Divers

- Plages ou intervalles de caractères et classes

Les plages de caractères dans les classes des expressions rationnelles (`/[a-z]/`) et dans l'opérateur `tr///` (aussi connu sous le nom `y///`) ne sont pas magiquement adaptées à Unicode. Cela signifie que `[A-Za-z]` ne correspondra

pas automatiquement à "toutes les lettres alphabétiques"; ce n'est déjà pas le cas pour les caractères 8 bits, et vous devrez utiliser `/[:alpha:]/` dans ce cas.

Pour spécifier une classe de caractères comme celle-ci dans les expressions rationnelles, vous pouvez utiliser plusieurs propriétés Unicode (`\pL`, ou éventuellement `\p{Alphabetic}`, dans ce cas particulier). Vous pouvez aussi utiliser un point de code Unicode comme borne de plage, mais il n'y a aucune magie associée à la spécification de ces plages. Pour plus d'informations, il existe des douzaines de classes de caractères Unicode, consultez *perlunicode*.

– Conversion de chaînes en nombres

Unicode définit plusieurs caractères séparateurs décimal et plusieurs caractères numériques, en dehors des chiffres habituels 0 à 9, tels que les chiffres Arabes et Indien. Perl ne supporte pas la conversion des chaînes vers des nombres pour les chiffres autres que 0 à 9 ASCII (et a à f ASCII pour l'hexadécimal).

### 49.1.13 Questions/Réponses

– Mes anciens scripts vont-ils encore fonctionner ?

Très probablement. À moins que vous ne génériez déjà des caractères Unicode vous-même, l'ancien comportement devrait être conservé. Le seul comportement qui change et qui pourrait générer de l'Unicode accidentellement est celui de `chr()` lorsqu'on lui fournit un argument supérieur à 255 : auparavant, il renvoyait le caractère correspondant à son argument modulo 255. Par exemple, `chr(300)` était égal à `chr(45)` ou "-" (en ASCII) alors que maintenant il produit LATIN CAPITAL LETTER I WITH BREVE.

– Comment rendre mes scripts compatibles avec Unicode ?

Très peu de choses doivent être faites sauf si vous génériez déjà des données Unicode. La plus importante est d'obtenir un flux d'entrée en Unicode ; reportez vous à la section ci-dessus concernant les entrées/sorties.

– Comment savoir si ma chaîne est en Unicode ?

Vous ne devriez pas vous en préoccuper. Non, vous ne devriez vraiment pas. Non, vraiment. Si vous le devez absolument, à part dans les cas décrits précédemment, ça signifie que vous n'avez pas encore atteint la transparence d'Unicode. Bon, puisque vous insistez :

```
print utf8::is_utf8($string) ? 1 : 0, "\n";
```

Mais notez bien que cela ne signifie pas qu'un caractère de la chaîne est forcément encodé en UTF-8, ou qu'un caractère a un point de code supérieur à 0xFF (255) ou même à 0x80 (128), ou que la chaîne contient au moins un caractère. La seule chose que `is_utf8()` fait, c'est retourner la valeur du drapeau interne "utf8ness" attaché à `$string`. Si ce drapeau n'est pas positionné, les octets du scalaire sont interprétés comme étant encodés sur un octet. S'il est positionné, les octets du scalaire sont interprétés comme les points de code (multi-octets, de taille variable) UTF-8 des caractères. Les octets ajoutés à une chaîne encodées en UTF-8 sont automatiquement convertis en UTF-8. Si un scalaire non-UTF-8 est combiné à un scalaire UTF-8 (par interpolation dans des guillemets, par concaténation explicite ou par substitution des paramètres de `printf/sprintf`), le résultat sera une chaîne encodée en UTF-8. Par exemple :

```
$a = "ab\x80c";
$b = "\x{100}";
print "$a = $b\n";
```

La chaîne affichée sera `ab\x80c = \x{100}\n` encodée en UTF-8, mais `$a` restera encodé en simple octet.

Il vous faudra quelque fois connaître la taille en octets d'une chaîne et non en caractères. Pour ceci utilisez la fonction `Encode::encode_utf8()` ou encore le pragma `bytes` et `length()`, la seule fonction qu'il définit :

```
my $unicode = chr(0x100);
print length($unicode), "\n"; # Affiche 1
require Encode;
print length(Encode::encode_utf8($unicode)), "\n"; # Affiche 2
use bytes;
print length($unicode), "\n"; # Affiche 2
(les codes 0xC4 0x80 de l'UTF-8)
```

– Comment puis-je détecter des données non valides pour un encodage particulier ?

Utilisez le package `Encode` pour tenter de les convertir. Par exemple :

```
use Encode 'decode_utf8';
if (decode_utf8($suite_octets_qu'on_pense_etre_utf8)) {
 # valide
} else {
 # invalide
}
```

Si il n'y a que UTF-8 qui vous intéresse, vous pouvez utiliser :

```
use warnings;
@chars = unpack("U0U*", $suite_octets_qu'on_pense_etre_utf8);
```

Si c'est invalide, un avertissement `Malformed UTF-8 character (byte 0x##) in unpack` est produit. "U0" signifie "n'attends strictement que de l'UTF-8". Sans ça, `unpack("U*",...)` accepterait des données telle que



`chr(0xFF)`, comme `pack` ainsi que nous l'avons vu précédemment.

- Comment convertir des données binaires en un encodage particulier et inversement ?

Ce n'est probablement pas aussi utile que vous pourriez le penser. Normalement, vous ne devriez pas en avoir besoin. Dans un sens, ce que vous demandez n'a pas beaucoup de sens : l'encodage concerne les caractères et des données binaires ne sont pas des "caractères", donc la conversion de "données" dans un encodage n'a pas sens à moins de connaître le jeu de caractères et l'encodage utilisés par ces données, mais dans ce cas il ne s'agit plus de données binaires.

Si vous disposez d'une séquence brute d'octets pour laquelle vous connaissez l'encodage particulier à utiliser, vous pouvez utiliser `Encode` :

```
use Encode 'from_to';
from_to($data, "iso-8859-1", "utf-8"); # de latin-1 à utf-8
```

L'appel à `from_to()` change les octets dans `$data`, mais la nature de la chaîne du point de vue de Perl n'est pas modifiée. Avant et après l'appel, la chaîne `$data` est juste un ensemble d'octets 8 bits. Pour Perl, l'encodage de cette chaîne reste "octets 8 bits natifs du système".

Vous pourriez mettre ceci en relation avec un hypothétique module 'Translate' :

```
use Translate;
my $phrase = "Yes";
Translate::from_to($phrase, 'english', 'deutsch');
phrase contient maintenant "Ja"
```

Le contenu de la chaîne change, mais pas la nature de la chaîne. Perl n'en sais pas plus après qu'avant sur la nature affirmative ou non de la chaîne.

Revenons à la conversion de données. Si vous avez (ou voulez) des données utilisant l'encodage 8 bits natif de votre système (c-à-d. Latin-1, EBCDIC, etc.), vous pouvez utiliser `pack/unpack` pour convertir vers/depuis Unicode.

```
$native_string = pack("C*", unpack("U*", $Unicode_string));
$Unicode_string = pack("U*", unpack("C*", $native_string));
```

Si vous avez une séquence d'octets que vous savez être de l'UTF-8 valide, mais que Perl ne le sait pas encore, vous pouvez le convaincre via :

```
use Encode 'decode_utf8';
$Unicode = decode_utf8($bytes);
```

Vous pouvez convertir de l'UTF-8 bien formé en une séquence d'octets, mais si vous voulez convertir des données binaires aléatoires en de l'UTF-8, c'est impossible. **Toutes les séquences d'octets ne forment par une chaîne UTF-8 bien formée.** Vous pouvez utiliser `unpack("C*", $string)` pour la première opération et vous pouvez créer une chaîne Unicode bien formée avec `pack("U*", 0xFF, ...)`.

- Comment puis-je afficher de l'Unicode ? Comment puis-je saisir du l'Unicode ?

Consultez <http://www.alanwood.net/unicode/> et <http://www.cl.cam.ac.uk/~mgk25/unicode.html>.

- Comment fonctionne Unicode avec les locales traditionnelles ?

En Perl, ça ne marche pas très bien. Evitez d'utiliser les locales avec le `pragma locale`. Utilisez soit l'un soit l'autre. Mais voyez `perlrun` pour la description de l'argument `-C` et la variable d'environnement équivalente `$ENV{PERL_UNICODE}` pour savoir comment activer les différentes fonctionnalités Unicode, par exemple en utilisant le paramétrage des locales.

#### 49.1.14 Notation hexadécimale

Le standard Unicode privilégie la notation hexadécimale car cela montre plus clairement la division d'Unicode en blocs de 256 caractères. L'hexadécimal est aussi plus court que le décimal. Vous pouvez toujours utiliser la notation décimale, bien sûr, mais l'apprentissage de l'hexadécimal vous facilitera la vie avec le standard Unicode. La notation `U+HHHH` utilise l'hexadécimal, par exemple.

Le préfixe `0x` indique un nombre hexadécimal, les chiffres étant `0` à `9` et `a` à `f` (ou `A` à `F`, la casse n'ayant pas d'importance). Chaque chiffre hexadécimal représente quatre bits, autrement dit la moitié d'un octet. `print 0x...`, `"\n"` affiche un nombre hexadécimal convertit en décimal et `printf "%x\n", $decimal` affiche un nombre décimal convertit en hexadécimal. Si vous n'avez que les "chiffres" d'un nombre hexadécimal, vous pouvez aussi utiliser la fonction `hex()`.

```
print 0x0009, "\n"; # 9
print 0x000a, "\n"; # 10
print 0x000f, "\n"; # 15
print 0x0010, "\n"; # 16
print 0x0011, "\n"; # 17
print 0x0100, "\n"; # 256

print 0x0041, "\n"; # 65
```

```
printf "%x\n", 65; # 41
printf "%#x\n", 65; # 0x41

print hex("41"), "\n"; # 65
```

### 49.1.15 Autres ressources

- Consortium Unicode  
  <http://www.unicode.org/>
- FAQ Unicode  
  <http://www.unicode.org/unicode/faq/>
- Glossaire Unicode  
  <http://www.unicode.org/glossary/>
- Unicode Useful Resources  
  <http://www.unicode.org/unicode/onlinedat/resources.html>
- Support Unicode et multilingue en HTML, fontes, navigateurs web et autres applications.  
  <http://www.alanwood.net/unicode/>
- FAQ UTF-8 et FAQ Unicode pour Unix/Linux  
  <http://www.cl.cam.ac.uk/~mgk25/unicode.html>
- Jeux de caractères traditionnels  
  <http://www.czyborra.com/>  
  <http://www.eki.ee/letter/>
- Les fichiers de support de l'Unicode de l'installation de Perl se trouvent dans le répertoire  
  `$Config{installprivlib}/unicore`  
  en Perl 5.8.0 ou plus récent, et dans  
  `$Config{installprivlib}/unicode`  
  dans les versions 5.6.x. (Le renommage en `lib/unicore` a été décidé pour éviter les conflits avec `lib/Unicode` sur les systèmes de fichiers insensible à la casse). Le fichier principale des données Unicode est `UnicodeData.txt` (or `Unicode.301` en Perl 5.6.1.) Vous pouvez obtenir la valeur de `$Config{installprivlib}` via  
  `perl "-V:installprivlib"`  
  Vous pouvez explorer les informations des fichiers de données Unicode en utilisant le module `Unicode::UCD`.

## 49.2 UNICODE DANS LES VERSIONS DE PERL PLUS ANCIENNES

Si vous ne pouvez pas mettre à jour votre Perl en version 5.8.0 ou plus récente, vous pouvez toujours traiter de l'Unicode en utilisant les modules `Unicode::String`, `Unicode::Map8`, et `Unicode::Map`, disponibles sur le CPAN. Si vous avez installé GNU recode, vous pouvez aussi utiliser le front-end Perl `Convert::Recode` pour les conversions de caractères.

Les exemples suivants proposent une conversion rapide d'octets ISO 8859-1 (Latin-1) en octets UTF-8 et inversement, ce code fonctionnant aussi avec les versions plus anciennes de Perl 5.

```
ISO 8859-1 vers UTF-8
s/([\x80-\xFF])/chr(0xC0|ord($1)>>6).chr(0x80|ord($1)&0x3F)/eg;

UTF-8 vers ISO 8859-1
s/([\xC2\xC3])([\x80-\xBF])/chr(ord($1)<<6&0xC0|ord($2)&0x3F)/eg;
```

## 49.3 REFERENCES

*perlunicode*, *Encode*, *encoding*, *open*, *utf8*, *bytes*, *perlretut*, *perlrun*, `Unicode::Collate`, `Unicode::Normalize` et `Unicode::UCD`.

## 49.4 REMERCIEMENTS

Merci aux lecteurs des listes de diffusion `perl5-porters@perl.org`, `perl-unicode@perl.org`, `linux-utf8@nl.linux.org` et `unicore@unicode.org` pour leurs commentaires précieux.

## 49.5 AUTEUR, COPYRIGHT, ET LICENCE

Copyright 2001-2002 Jarkko Hietaniemi <jhi@iki.fi>

Ce document peut être distribué sous les mêmes conditions que Perl lui-même.

## 49.6 TRADUCTION

### 49.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 49.6.2 Traducteur

Marc Carmier <mcarmier@gmail.com>.

### 49.6.3 Relecture

Paul Gaborit (Paul.Gaborit arobase enstimac.fr).

# Chapitre 50

## perlsec

Sécurité de Perl

### 50.1 DESCRIPTION

Perl est conçu pour faciliter une programmation sûre, même lorsqu'il tourne avec des privilèges spéciaux, comme pour les programmes `setuid` ou `setgid`. Contrairement à la plupart des shells de ligne de commande, qui sont basés sur de multiples passes de substitution pour chaque ligne du script, Perl utilise un procédé d'évaluation plus conventionnel contenant moins de pièges cachés. De plus, comme le langage a plus de fonctionnalités intégrées, il doit moins se reposer sur des programmes externes (et potentiellement peu sûrs) pour accomplir ses tâches.

Perl met en oeuvre automatiquement un ensemble de vérifications spécifiques à la sécurité, appelé *taint mode* (mode souillé, NDT), lorsqu'il détecte que son programme tourne avec des identifiants de groupe ou d'utilisateurs réel et effectif différents. Le bit `setuid` dans les permissions d'Unix est le mode 04000, le bit `setgid` est le mode 02000 ; ils peuvent être placés l'un ou l'autre, ou les deux à la fois. Vous pouvez aussi activer le taint mode explicitement en utilisant l'option de ligne de commande `-T`. Cette option est *fortement* conseillée pour les programmes serveurs et pour tout programme exécuté au nom de quelqu'un d'autre, comme un script CGI. Une fois que le taint mode est activé, il l'est pour tout le reste de votre script.

Lorsqu'il est dans ce mode, Perl prend des précautions spéciales appelées *taint checks* (vérification de pollution, NDT) pour éviter aussi bien les pièges évidents que les pièges subtils. Certaines de ces vérifications sont raisonnablement simples, comme vérifier que personne ne peut écrire dans les répertoires du `path` ; les programmeurs précautionneux ont toujours utilisé de telles méthodes. D'autres vérifications, toutefois, sont mieux supportées par le langage lui-même, et ce sont ces vérifications en particulier qui contribuent à rendre un programme Perl `set-id` plus sûr qu'un programme équivalent en C.

Vous ne pouvez pas utiliser des données provenant de l'extérieur de votre programme pour modifier quelque chose d'autre à l'extérieur – au moins pas par accident. Tous les arguments de ligne de commande, toutes les variables d'environnement, toutes les informations locales (voir *perllocale*), résultant de certains appels au système (`readdir()`, `readlink()`, la variable de `shmread()`, les messages renvoyés par `msgrcv()`, le mot de passe, les champs `gecos` et `shell` des appels `getpwnxx()`, et toutes les entrées par fichier sont marquées comme "souillées". Les données souillées ne peuvent pas être utilisées directement ou indirectement dans une commande qui invoque un sous-shell, ni dans toute commande qui modifie des fichiers, des répertoires ou des processus, **avec les exceptions suivantes** :

- Les fonctions `print` et `syswrite` **ne** vérifient **pas** si leurs arguments sont souillées ou non.
- La sûreté des valeurs utilisées comme méthodes symboliques

```
$obj->$method(@args);
```

ou comme références symboliques à des sous-programmes

```
&{$foo}(@args);
```

```
$foo->(@args);
```

n'est pas vérifiée. Cela nécessite une attention particulière à moins que vous acceptiez que des données externes affectent votre flux de contrôle. Si vous ne limitez pas très précisément ce que seront ces valeurs symboliques, les personnes les fournissant pourront appeler des fonctions **en dehors** de votre code Perl, telle que `POSIX::system`, et seront donc à même d'exécuter n'importe quel code externe.

- Les clés des tables de hachage ne sont **jamais** souillées.

Pour des raisons d'efficacité, Perl a un point de vue très conservateur sur ce qui est souillé ou non. Si une expression contient des données souillées, n'importe quelle sous-expression sera considérée comme souillée, même si la valeur de cette sous-expression n'est pas affectée par des données souillées.

Puisque la sûreté est associée à chaque valeur scalaire, certains éléments d'un tableau peuvent être souillés et d'autres pas. Les clés d'une table de hachage ne sont **jamais** souillées.

Par exemple :

```

$arg = shift; # $arg est souillée
$hid = $arg, 'bar'; # $hid est aussi souillée
$line = <>; # Souillée
$line = <STDIN>; # Souillée aussi
open FOO, "/home/me/bar" or die $!;
$line = <FOO>; # Encore souillée
$path = $ENV{'PATH'}; # Souillée, mais voir plus bas
$data = 'abc'; # Non souillée

system "echo $arg"; # Non sûr
system "/bin/echo", $arg; # Considéré comme non sûr
 # (Perl ne sait rien de /bin/echo)
system "echo $hid"; # Non sûr
system "echo $data"; # Non sûr jusqu'à ce que PATH soit fixé

$path = $ENV{'PATH'}; # $path est désormais souillée

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'}; # $path N'est maintenant PLUS souillée
system "echo $data"; # Est désormais sûr !

open(FOO, "< $arg"); # ok - fichier en lecture seule
open(FOO, "> $arg"); # Pas ok - tentative d'écriture

open(FOO,"echo $arg|"); # Pas ok
open(FOO,"-|")
 or exec 'echo', $arg; # Pas ok non plus

$shout = 'echo $arg'; # Non sûr, $shout est maintenant souillée

unlink $data, $arg; # Non sûr
umask $arg; # Non sûr

exec "echo $arg"; # Non sûr
exec "echo", $arg; # Non sûr
exec "sh", '-c', $arg; # Extrêmement non sûr !

@files = <*.c>; # Non sûr (utilise readdir() ou équivalent)
@files = glob('*.c'); # Non sûr (utilise readdir() ou équivalent)

Les versions de Perl antérieures à 5.6.0 utilisaient un programme
externe pour trouver les noms de fichiers dans <*.c> ou glob('*.c').
Mais dans tous les cas, le résultat est souillé puisque la liste des
noms de fichiers provient de l'extérieur du programme.

$bad = ($arg, 23); # $bad est souillé
$arg, 'true'; # Non sûr (même si ça ne l'est pas)

```

Si vous essayez de faire quelque chose qui n'est pas sûr, vous obtiendrez une erreur fatale disant quelque chose comme "Insecure dependency" ou "Insecure \$ENV{PATH}".

L'exception à la règle "une valeur souillée souille l'ensemble de l'expression" est l'opérateur ternaire conditionnel ?: . Puisque le code utilisant l'opérateur conditionnel :

```
$result = $valeur_souillee ? "Pas souillé" : "Pas souillé non plus";
```

est en fait :

```
if ($valeur_souillee) {
 $result = "Pas souillé";
} else {
 $result = "Pas souillé non plus";
}
```

cela n'aurait pas beaucoup de sens de souillé \$result.

### 50.1.1 Blanchiment et détection des données souillées

Pour tester si une variable contient des données souillées, et quels usages provoqueraient ainsi un message "Insecure dependency", vous pouvez utiliser la fonction `tainted()` du module `Scalar::Util`, disponible sur CPAN ou inclus dans Perl depuis la version 5.8.0. Ou vous pouvez utiliser la fonction `is_tainted` suivante :

```
sub is_tainted {
 return ! eval { eval("#" . substr(join('', @_), 0, 0)); 1; };
}
```

Cette fonction utilise le fait que la présence de données souillées n'importe où dans une expression rend toute l'expression souillée. Il serait inefficace de tester la sûreté de tous les arguments pour tous les opérateurs. Au lieu de cela, l'approche légèrement plus efficace et conservatrice qui est utilisée est que si une valeur souillée a été accédée à l'intérieur d'une expression, alors la totalité de l'expression est considérée comme souillée.

Mais le test de pureté ne vous fournit rien d'autre. Parfois, vous devez juste rendre vos données propres. Une valeur peut être blanchie en l'utilisant comme clé de table de hachage. Sinon, la seule façon d'outrepasser le mécanisme de pollution est de référencer des sous-motifs depuis une expression régulière. Perl présume que si vous référencez une sous-chaîne en utilisant \$1, \$2, etc., c'est que vous saviez ce que vous étiez en train de faire lorsque vous rédigez le motif. Cela implique un peu de réflexion – ne blanchissez pas tout aveuglément, ou vous détruisez la totalité du mécanisme. Il est meilleur de vérifier que la variable ne contient que des bons caractères (pour certaines valeurs de "bon") plutôt que de vérifier qu'elle contient un quelconque mauvais caractère. C'est parce qu'il est beaucoup trop facile de manquer un mauvais caractère auquel vous n'avez jamais pensé.

Voici un test pour s'assurer que les données ne contiennent rien d'autre que des caractères de "mots" (alphabétiques, numériques et souligné), un tiret, une arobase, ou un point.

```
if ($data =~ /^[^\@\w.]+$/) {
 $data = $1; # $data est maintenant propre
} else {
 die "Bad data in '$data'"; # tracer cela quelque part
}
```

Ceci est assez sûr car `/\w+/` ne correspond normalement pas aux métacaractères du shell, et les points, tirets ou arobases ne veulent rien dire de spécial pour le shell. L'usage de `./+`  n'aurait pas été sûr en théorie parce qu'il laisse tout passer, mais Perl ne vérifie pas cela. La leçon est que lorsque vous blanchissez, vous devez être excessivement précautionneux avec vos motifs. Le blanchiment des données à l'aide d'expressions régulières est le *seul* mécanisme pour nettoyer les données polluées, à moins que vous n'utilisiez la stratégie détaillée ci-dessous pour forker un fils ayant des privilèges plus faibles.

L'exemple ne nettoie pas \$data si `use locale` est en cours d'utilisation, car les caractères auxquels correspond `\w` sont déterminés par la localisation. Perl considère que les définitions locales ne sont pas sûres car elles contiennent des données extérieures au programme. Si vous écrivez un programme conscient de la localisation, et si voulez blanchir des données avec une expression régulière contenant `\w`, mettez `no locale` avant l'expression dans le même bloc. Voir **SÉCURITÉ** in *perllocale* pour plus de précisions et des exemples.

### 50.1.2 Options sur la ligne "#!"

Quand vous rendez un script exécutable, de façon à pouvoir l'utiliser comme une commande, le système passera des options à perl à partir de la ligne `#!` du script. Perl vérifie que toutes les options de ligne de commande données à un script `setuid` (ou `setgid`) correspondent effectivement à celles placées sur la ligne `#!`. Certains Unix et environnements cousins imposent une limite d'une seule option sur la ligne `#!`, vous aurez donc peut-être besoin d'utiliser quelque chose comme `-wU` à la place de `-w -U` sous ces systèmes (ce problème ne devrait se poser qu'avec les Unix et les environnements proches qui supportent `#!` et les scripts `setuid` ou `setgid`).

### 50.1.3 Mode taint et @INC

Lorsque le mode "taint" (-T) est actif, le répertoire "." ne fait plus partie de @INC et les variables d'environnement PERL5LIB et PERLLIB sont ignorées par Perl. Vous pouvez encore modifier @INC depuis l'extérieur du programme en utilisant l'option -I de la ligne de commande comme expliqué dans *perlrun*. Les deux variables d'environnement sont ignorées parce qu'elles sont cachées et qu'un utilisateur qui lance un programme ne sait pas obligatoirement qu'elles existent alors que l'option -I est clairement visible et peut donc être tolérée.

Un autre moyen de modifier @INC sans modifier le programme consiste à utiliser la directive `lib` de la manière suivante :

```
perl -Mlib=/foo programme
```

L'avantage de `-Mlib=/foo` sur `-I/foo` est de gérer les éventuelles occurrences multiples d'un même répertoire (le répertoire ne sera pas répété dans @INC).

Notez que si un chemin souillé est ajouté à @INC, le problème suivant sera détecté :

```
Insecure dependency in require while running with -T switch
```

### 50.1.4 Nettoyer votre PATH

Pour les messages "Insecure \$ENV{PATH}", vous avez besoin de fixer `$ENV{'PATH'}` à une valeur connue, et chaque répertoire dans le PATH doit être spécifié sous la forme d'un chemin absolu et non modifiable par d'autres utilisateurs que son propriétaire et son groupe. Vous pourriez être surpris d'obtenir ce message alors même que le chemin vers votre exécutable est un chemin absolu. Ceci n'est *pas* généré parce que vous n'avez pas fourni un chemin complet au programme mais plutôt parce que vous n'avez jamais défini votre variable d'environnement PATH, ou vous ne l'avez pas défini comme quelque chose de sûr. Puisque Perl ne peut pas garantir que l'exécutable en question ne vas pas exécuter un autre programme qui dépend de votre PATH, il s'assure que vous avez défini le PATH.

Le PATH n'est pas la seule variable d'environnement qui peut poser des problèmes. Puisque certains shells peuvent utiliser les variables IFS, CDPATH, ENV, et BASH\_ENV, Perl vérifie que celles-ci sont soit vides, soit propres, lorsqu'il démarre des sous- processus. Vous pourriez désirer ajouter quelque chose comme ceci à vos scripts `setid` et `blanchisseurs`.

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)}; # Rend %ENV plus sûr
```

Il est aussi possible de s'attirer des ennuis avec d'autres opérations qui ne se soucient pas si elles utilisent des valeurs souillées. Faites un usage judicieux des tests de fichiers quand vous avez affaire à un nom de fichier fournis par l'utilisateur. Lorsque c'est possible, réalisez les ouvertures et compagnie **après** avoir correctement abandonné les privilèges d'utilisateur (ou de groupe !) particuliers. Perl ne vous empêche pas d'ouvrir en lecture des noms de fichier souillés, alors faites attention à ce que vous imprimez. Le mécanisme de pollution est destiné à prévenir les erreurs stupides, pas à supprimer le besoin de réflexion.

Perl n'appelle pas de shell pour expander les métacaractères (wildcards, NDT) quand vous passez des listes de paramètres explicites à `system` et `exec` au lieu de chaînes pouvant contenir des métacaractères. Malheureusement, les fonctions `open`, `glob`, et `backtick` (substitution de commande avec "", NDT) ne fournissent pas de telles conventions d'appel alternatives, alors d'autres subterfuges sont requis.

Perl fournit une façon raisonnablement sûre d'ouvrir un fichier ou un tube depuis un programme `setuid` ou `setgid` : créez juste un processus fils ayant des privilèges réduits et qui fera le sale boulot pour vous. Tout d'abord, créez un fils en utilisant la syntaxe spéciale de `open` qui connecte le père et le fils par un tube. Puis le fils redéfinit son ensemble d'ID et tous les autres attributs dépendant du processus, comme les variables d'environnement, les umasks, les répertoires courants, pour retourner à des valeurs originelles ou connues comme sûres. Puis le processus fils, qui n'a plus la moindre permission spéciale, réalise le `open` ou d'autres appels système. Finalement, le fils passe à son père les données auxquelles il parvient à accéder. Puisque le fichier ou le tube ont été ouverts par le fils alors qu'il tournait avec des privilèges inférieurs à celui du père, il ne peut pas être trompé et faire quelque chose qu'il ne devrait pas.

Voici une façon de faire des substitutions de commande de façon raisonnablement sûre. Remarquez comment le `exec` n'est pas appelé avec une chaîne que le shell pourrait interpréter. C'est de loin la meilleure manière d'appeler quelque chose qui pourrait être sujet à des séquences d'échappements du shell : n'appellez tout simplement jamais le shell.

```
use English '-no_match_vars';
die "Can't fork: $!" unless defined($pid = open(KID, "-|"));
if ($pid) { # parent
 while (<KID>) {
```

```

 # faire quelque chose
}
close KID;
} else {
my @temp = ($EUID, $EGID);
my $orig_uid = $UID;
my $orig_gid = $GID;
$EUID = $UID;
$EGID = $GID;
suppression des privilèges
$UID = $orig_uid;
$GID = $orig_gid;
S'assurer que les privilèges sont réellement partis
($EUID, $EGID) = @temp;
die "Can't drop privileges"
 unless $UID == $EUID && $GID eq $EGID;
$ENV{PATH} = "/bin:/usr/bin"; # PATH minimaliste
exec 'myprog', 'arg1', 'arg2'
 or die "can't exec myprog: $!";
}

```

Une stratégie similaire marcherait pour l'expansion des métacaractères via `glob`, bien que vous pouvez utiliser `readdir` à la place.

La vérification de la sûreté est surtout utile lorsque, même si vous vous faites confiance de ne pas avoir écrit un programme ouvrant toutes les portes, vous ne faites pas nécessairement confiance à ceux qui finiront par l'utiliser et pourraient essayer de le tromper pour qu'il fasse de vilaines choses. C'est le genre de vérification de sécurité qui est utile pour les programmes `set-id` et les programmes qui sont lancés au nom de quelqu'un d'autre, comme les scripts CGI.

C'est très différent, toutefois, du cas où l'on ne fait même pas confiance à l'auteur du code de ne pas essayer de faire quelque chose de diabolique. Ceci est le type de confiance dont on a besoin quand quelqu'un vous tend un programme que vous n'avez jamais vu auparavant et vous dit : "Voilà, exécute ceci". Pour ce genre de sécurité, jetez un oeil au module `Safe`, inclu en standard dans la distribution de Perl. Ce module permet au programmeur de mettre en place des compartiments spéciaux dans lesquels toutes les opérations liées au système sont détournées et où l'accès à l'espace de noms est contrôlé avec soin.

### 50.1.5 Failles de sécurité

Au-delà des problèmes évidents qui découlent du fait de donner des privilèges spéciaux à des systèmes aussi flexibles que les scripts, sous de nombreuses versions d'Unix, les scripts `set-id` ne sont pas sûrs dès le départ de façon inhérente. Le problème est une race condition dans le noyau. Entre le moment où le noyau ouvre le fichier pour voir quel interpréteur il doit exécuter et celui où l'interpréteur (maintenant `set-id`) se retourne et réouvre le fichier pour l'interpréter, le fichier en question peut avoir changé, en particulier si vous avez des liens symboliques dans votre système.

Heureusement, cette "caractéristique" du noyau peut parfois être invalidée. Malheureusement, il y a deux façons de l'invalider. Le système peut simplement déclarer hors-la-loi les scripts ayant un bit `set-id` mis, ce qui n'aide pas vraiment. Sinon, il peut simplement ignorer les bits `setuid` pour les scripts. Si ce dernier cas est vrai, Perl peut émuler le mécanisme `setuid` et `setgid` lorsqu'il remarque les bits `setuid/gid` bits, par ailleurs inutiles, sur des scripts Perl. Il le fait via un exécutable spécial appelé *suidperl* qui est appelé automatiquement pour vous si besoin est.

Toutefois, si la caractéristique du noyau pour les scripts `set-id` n'est pas invalidée, Perl se plaindra bruyamment que votre script `set-id` n'est pas sûr. Vous devrez soit invalider la caractéristique du noyau pour les scripts `set-id`, soit mettre un wrapper C autour du script. Un wrapper C est juste un programme compilé qui ne fait rien à part appeler votre programme Perl. Les programmes compilés ne sont pas sujets au bug du noyau qui tourmente les scripts `set-id`. Voici un wrapper simple, écrit en C :

```

#define REAL_PATH "/path/to/script"
main(ac, av)
 char **av;
{
 execl(REAL_PATH, av);
}

```



Compilez ce wrapper en un binaire exécutable, puis rendez-*it* setuid ou setgid à la place de votre script.

Ces dernières années, les vendeurs ont commencé à fournir des systèmes libérés de ce bug de sécurité inhérent. Sur de tels systèmes, lorsque le noyau passe le nom du script set-id à ouvrir à l'interpréteur, plutôt que d'utiliser un nom et un chemin sujets à l'ingérence, il passe */dev/fd/3*. C'est un fichier spécial déjà ouvert sur le script, de sorte qu'il ne peut plus y avoir de race condition que des scripts malins pourraient exploiter. Sur ces systèmes, Perl devrait être compilé avec l'option `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. Le programme *Configure* qui construit Perl essaye de trouver cela tout seul, vous ne devriez donc jamais avoir à spécifier cela vous-même. La plupart des versions modernes de SysVr4 et BSD 4.4 utilisent cette approche pour éviter la race condition du noyau.

Avant la version 5.6.1 de Perl, des bugs dans le code de **suidperl** pouvait introduire des failles de sécurité.

### 50.1.6 Protection de vos programmes

Il existe de nombreuses façons de cacher la source de vos programmes Perl, avec des niveaux variables de "sécurité".

Tout d'abord, toutefois, vous *ne pouvez pas* retirer la permission en lecture, car le code source doit être lisible de façon à être compilé et interprété (cela ne veut toutefois pas dire que le source d'un script CGI est lisible par n'importe qui sur le web). Vous devez donc laisser les permissions au niveau socialement amical de 0755. Ceci laisse voir votre source uniquement aux gens de votre système local.

Certaines personnes prennent par erreur ceci pour un problème de sécurité. Si votre programme fait des choses qui ne sont pas sûres, et s'appuie sur le fait que les gens ne savent pas comment exploiter ces failles, il n'est pas sûr. Il est souvent possible pour quelqu'un de déterminer les failles de sécurité et de les exploiter sans voir le source. La sécurité par l'obscurité, expression désignant le fait de cacher vos bugs au lieu de les corriger, est vraiment une faible sécurité.

Vous pouvez essayer d'utiliser le chiffrement via des filtres de sources (Filter::*\** sur CPAN ou Filter::*Util::Call* et Filter::*Simple* depuis Perl 5.8). Mais les craqueurs peuvent encore le déchiffrer. Vous pouvez essayer d'utiliser le compilateur de byte code et l'interpréteur décrit ci-dessous, mais les craqueurs peuvent encore le décompiler. Vous pouvez essayer d'utiliser le compilateur de code natif décrit ci-dessous, mais les craqueurs peuvent encore le désassembler. Ces solutions posent des degrés variés de difficulté aux gens voulant obtenir votre code, mais aucune ne peut définitivement le dissimuler (ceci est vrai pour tous les langages, pas uniquement Perl).

Si vous êtes inquiet que des gens profitent de votre code, alors le point crucial est que rien à part une licence restrictive ne vous donnera de sécurité légale. Licenciez votre logiciel et pimentez-le de phrases menaçantes comme "Ceci est un logiciel propriétaire non publié de la société XYZ. L'accès qui vous y est donné ne vous donne pas la permission de l'utiliser bla bla bla". Vous devriez voir un avocat pour être sûr que votre vocabulaire tiendra au tribunal.

### 50.1.7 Unicode

Unicode est une technologie nouvelle et complexe qui peut facilement amener quelqu'un vers de nouvelles embûches liées à la sécurité. Voir *perluniintro* pour la présentation générale, *perlunicode* pour les détails et en particulier `perl<perlunicode/"Implications d'Unicode sur la sécurité">`.

### 50.1.8 Attaques par complexité algorithmique

Certains algorithmes internes utilisés dans l'implémentation de Perl peuvent être attaqués en fournissant des entrées choisies spécialement pour consommer soit beaucoup de temps, soit beaucoup de mémoire, soit les deux. C'est ce qu'on appelle des attaques par *Déni de service* (DoS - Denial of Service).

– La fonction de hachage - l'algorithme utilisé pour "ordonné" les clés des tables de hachage a changé plusieurs fois au cours du développement de Perl, principalement pour des raisons d'efficacité. En Perl 5.8.1, les aspects concernant la sécurité ont aussi été pris en compte.

Dans les versions antérieures à la 5.8.1, il était relativement facile de générer des données qui, en tant que clés de hachage, amenait Perl à consommer énormément de temps à cause d'une mauvaise gestion des structures internes de hachage. En Perl 5.8.1, la fonction de hachage est volontairement perturbée par un générateur pseudo-aléatoire afin de rendre très difficile la génération de mauvaises clés de hachage. Voir `PERL_HASH_SEED` in *perlrun* pour plus d'information.

La perturbation aléatoire est utilisée par défaut mais si quelqu'un souhaite simuler l'ancien comportement, il peut positionner la variable d'environnement `PERL_HASH_SEED` à zéro (ou toute autre valeur entière). Une raison pour vouloir retrouver cet ancien comportement est qu'avec le nouveau comportement, deux exécutions successives de Perl ordonnent les clés de hachage différemment ce qui peut perturber des applications (avec `Data::Dumper`, les sorties de deux exécutions sur les mêmes données ne sont pas identiques).

**Perl n'a jamais garanti l'ordre des clés des tables de hachage** et cet ordre a déjà changé plusieurs fois depuis l'arrivée de Perl 5. De plus, l'ordre des clés a toujours été et continue d'être modifié par une insertion.

Notez aussi que bien que cet ordre puisse être considéré comme aléatoire, il ne devrait pas **pas** être utilisé pour du mélange aléatoire de liste (utilisez la fonction `List::Util::shuffle()` du module `List::Util` qui est un module standard depuis Perl 5.8.0 ou le module CPAN `Algorithm::Numerical::Shuffle`), pour générer des permutations (utilisez les modules CPAN `Algorithm::Permute` ou `Algorithm::FastPermute`) ou pour des applications de cryptographie.

- Les expressions rationnelles - Le moteur d'expression rationnelle de Perl est aussi appelé NFA (Non-deterministic Finite Automaton - Automate fini non déterministe) ce qui signifie, entre autres, qu'il peut facilement consommer énormément de temps et d'espace mémoire si une expression rationnelle est reconnaissable de nombreuses manières différentes. Des expressions rationnelles correctement conçues peuvent empêcher cela mais c'est loin d'être facile (nous vous recommandons la lecture du livre "Mastering Regular Expressions", voir aussi *perlfaq2*). L'utilisation d'espace mémoire trop important se manifeste par un dépassement de capacité de mémoire (out of memory) de Perl.
- Le tri - l'algorithme de tri rapide (quicksort) utilisé dans les versions de Perl antérieures à la version 5.8.0 pour implémenter la fonction `sort()` est facilement perturbable pour l'amener à consommer beaucoup de temps. Le simple tri d'une liste déjà triée suffit. Depuis la version 5.8.0 de Perl, un autre algorithme de tri est utilisé : le 'mergesort'. Ce nouvel algorithme est insensible à ses données d'entrée. Il ne peut donc plus être dupé.

Pour plus d'information voir <http://www.cs.rice.edu/~scrosby/hash/> et n'importe quel livre d'informatique abordant la complexité algorithmique.

## 50.2 VOIR AUSSI

*perlrun* pour sa description du nettoyage des variables d'environnement.

## 50.3 TRADUCTION

### 50.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 50.3.2 Traducteur

Traduction initiale : Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Mise à jour : Paul Gaborit <[paul.gaborit@enstimac.fr](mailto:paul.gaborit@enstimac.fr)>.

### 50.3.3 Relecture

Régis Julié <[regis.julie@cetelem.fr](mailto:regis.julie@cetelem.fr)>

# Chapitre 51

## perlmod

Modules Perl (paquetages et tables de symboles)

### 51.1 DESCRIPTION

#### 51.1.1 Paquetages

Perl fournit un mécanisme de namespace alternatif pour éviter aux paquetages de s'écraser mutuellement les variables. En fait, il n'y a rien qui ressemble aux variables globales en perl (bien que quelques identificateurs appartiennent par défaut au paquetage principal plutôt qu'au paquetage courant). L'instruction `package` déclare l'unité de compilation comme étant le namespace utilisé. La visibilité d'une déclaration de paquetage est de la déclaration jusqu'à la fin du bloc, `eval`, `sub`, ou la fin du fichier (c'est la même visibilité que les opérateurs `my()` et `local()`). Tous les identificateurs dynamiques suivants seront dans le même namespace. L'instruction `package` affecte uniquement les variables dynamiques – ainsi que celles sur lesquelles vous avez utilisé `local()` – mais *pas* sur les variables lexicales créées à l'aide de `my()`. Typiquement, cela serait la première déclaration dans un fichier à être incluse par un `require` ou un `use`. Vous pouvez inclure un paquetage dans plusieurs endroits; cela n'a quasiment aucune influence sur la table de symboles utilisée par le compilateur pour le reste du bloc. Vous pouvez utiliser les variables et les fichiers d'autres paquetages en préfixant l'identificateur avec le nom du paquetage et d'un double ":" : `$Package::Variable`. Si le nom de paquetage est nul, le `main` est utilisé. Donc, `$$::sail` est équivalent à `$main::sail`.

L'ancien délimiteur de paquetage était une apostrophe, mais un double ":" est maintenant utilisé, parce que c'est plus facilement compréhensible par les humains, et parce que c'est plus pratique pour les macros d'**emacs**. Cela fait aussi croire aux programmeurs C++ qu'ils comprennent ce qui se passe – en opposition à l'apostrophe qui faisait penser aux programmeurs Ada qu'ils comprenaient ce qui se passait. Comme l'ancienne méthode est toujours supportée pour préserver la compatibilité ascendante, si vous essayez une chaîne comme "This is \$owner's house", vous allez en fait accéder à `$owner::s`; c'est à dire, la variable `$s` du paquetage `owner`, ce qui n'est probablement pas ce que vous vouliez. Utilisez des accolades pour supprimer l'ambiguïté, comme ça : "This is \${owner}'s house".

Les paquetages peuvent être imbriqués dans d'autres paquetages : `$EXTERNE::INTERNE::variable`. D'ailleurs, ceci n'implique rien dans l'ordre de recherche des noms. Tous les symboles sont soit locaux dans le paquetage courant, soit doivent avoir leur nom complet. Par exemple, dans le paquetage `EXTERNE`, `$INTERNET::var` ne se réfère pas à `$EXTERNE::INTERNE::var`. Il croira que le paquetage `INTERNE` est un paquetage totalement séparé.

Seuls les identificateurs commençant par une lettre (ou un underscore) sont stockés dans la table de symboles des paquetages. Tous les autres symboles sont gardés dans le paquetage `main`, ceci inclut les variables de ponctuations telles `$_`. De plus, les identificateurs `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC`, et `SIG` sont stockés dans le paquetage `main` lorsque ils ne sont pas redéfinis, même si ils sont utilisés dans un autre but que celui pour lequel ils ont été créés. Notez aussi que si vous avez un paquetage appelé `m`, `s`, ou `y`, alors, vous ne pourrez pas utiliser la forme qualifiée d'un identificateur, car il sera interprété comme un patron de recherche, substitution, ou remplacement.

(Les variables qui commencent par un underscore étaient à l'origine dans le paquetage `main`, mais nous avons décidé qu'il serait plus utile aux programmeurs de paquetages de préfixer leurs variables locales et noms de méthodes avec un underscore. `$_` est bien sûr toujours global.)

Les chaînes qui sont utilisés avec `eval()` sont compilées dans le paquetage ou l'`eval()` à été compilé. (Les assignements à `$$SIG{}`, d'un autre côté, supposent que le signal spécifié est dans le paquetage `main`. Mais vous pouvez dire au signal d'être dans le paquetage.) Par exemple, examinez `perldb.pl` dans la librairie Perl. Il passe dans le paquetage `DB` pour éviter que le débogueur n'interfère avec les variables du script que vous essayez de déboguer. De temps en temps, il revient

au paquetage `main` pour évaluer différentes expressions dans le contexte du paquetage `main` (ou de la ou vous venez). Reférez vous à *perldebug*.

Le symbole spécial `__PACKAGE__` contient le paquetage courant, mais ne peut pas être (facilement) utilisé pour construire des variables.

Référez vous à *perlsub* pour de plus amples informations sur `my()` et `local()`, et *perlref* pour les détails.

### 51.1.2 Tables de symboles

Les tables de symboles pour un paquetage sont stockées dans un hash du même nom avec deux ":" à la fin. La table de symbole de `main` est donc `%main::`, ou `%::` pour raccourcir. De même, la table de symbole d'un paquetage imbriqué est nommée `%EXTERNE::INTERNE::`.

La valeur de chaque entrée du hash est ce à quoi vous vous référez quand vous utilisez la notation `*name`. En fait, les deux instructions suivantes ont le même effet, bien que la première soit plus efficace car il y a une vérification lors de la compilation :

```
local *main::truc = *main::machin;
local $main::{truc} = $main::{machin};
```

Vous pouvez les utiliser pour imprimer toutes les variables d'un paquetage, par exemple, la librairie standard *dumpvar.pl* et le module CPAN `Devel::Symdump` l'utilisent.

L'assignement à un typeglob crée juste un alias, par exemple :

```
*dick = *richard;
```

font que les variables, sous fonctions, formats, et noms de fichiers et de répertoires accessibles via l'identificateur `richard` aussi accessible via l'identificateur `dick`. Si vous voulez juste faire un alias d'une variable, ou d'une sous fonction, vous devez assigner une référence à la place :

```
*dick = \$richard;
```

Ce qui fait de `$richard` et `$dick` la même chose, mais laisse les tableaux `@richard` et `@dick` différents. Pas mal hein ?

Ce mécanisme peut être utilisé pour passer et retourner des références vers ou depuis une sous fonction si vous ne voulez pas copier l'ensemble. Cela fonctionne uniquement avec les variables dynamiques, pas les lexicales.

```
%un_hash = (); # ne peut pas être my()
*un_hash = fn(\%un_autre_hash);
sub fn {
 local *hash_pareil = shift;
 # maintenant, utilisez %hashsym normalement,
 # et vous changerez aussi le %un_autre_hash
 my %nhash = (); # Faites ce que vous voulez
 return \%nhash;
}
```

Au retours, la référence écrasera le hash dans la table de symboles spécifié par le typeglob `*un_hash`. Ceci est une manière rapide de jouer avec les références quand vous ne voulez pas avoir à déréférencer des variables explicitement.

Une autre utilisation des tables de symboles est d'avoir des variables "constantes".

```
*PI = \3.14159265358979;
```

Maintenant, vous ne pouvez plus modifier `$PI`, ce qui est une bonne chose après tout. Ceci n'est pas la même chose qu'une sous fonction constante, qui est sujette à des optimisations lors de la compilation. Ce n'est pas la même chose. Une sous fonction constante est une qui ne prends pas d'arguments, et retourne une expression constante. Référez vous à *perlsub* pour plus de détails. Le pragma `use constant` est un truc pratique pour ce genre de choses.

Vous pouvez dire `*foo{PACKAGE}` et `*foo{NAME}` pour trouver de quels noms et paquetages le symbole `*foo` provient. Ceci peut être utile dans une fonction qui reçoit des typeglob comme arguments :

```

sub identify_typeglob {
 my $glob = shift;
 print 'Vous m\'avez donné ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME}, "\n";
}
identify_typeglob *foo;
identify_typeglob *bar::baz;

```

Ceci imprimera

```

Vous m\'avez donné main::foo
Vous m\'avez donné bar::baz

```

La notation `*foo{THING}` peut aussi être utilisé pour obtenir une référence a des éléments de `*foo`. Référez vous à *perlref*.

### 51.1.3 Constructeurs et Destructeurs de paquetage

Il y a deux définitions de fonctions spéciales qui servent de constructeur et de destructeur de paquetage. Elles sont `BEGIN` et `END`. Le `sub` est optionnel pour ces deux routines.

Une fonction `BEGIN` est exécutée des que possible, c'est a dire, le moment ou le paquetage est complètement défini, avant que le reste du fichier soit parsé. Vous pouvez avoir plusieurs blocs `BEGIN` dans un fichier – ils seront exécutés dans l'ordre d'apparition. Parce que un bloc `BEGIN` s'exécute immédiatement, il peut définir des sous fonctions ainsi que pas mal de choses depuis d'autres fichiers pour les rendre visibles depuis le reste du fichier. Dès qu'un `BEGIN` a été exécuté, toutes les ressources qu'il utilisait sont détruites et sont rendues a Perl. Cela signifie que vous ne pouvez pas appeler explicitement un `BEGIN`.

Une fonction `END` est exécutée aussi tard que possible, c'est a dire, quand l'interpréteur se termine, même si sa sortie est due a un appel à `die()`. (Mais pas si il se relance dans un autre via `exec`, ou est terminé par un signal – vous aurez a gérer ça vous même (si c'est possible).) Vous pouvez avoir plein de blocs `END` dans un fichier – ils seront exécutés dans l'ordre inverse de leur définition, c'est à dire le dernier d'abord (last in, first out (LIFO)).

Au sein d'une fonction `END`, `$?` contient la valeur que le script va passer a `exit()`. vous pouvez modifier `$?` pour changer la valeur de sortie du script. Attention a ne pas changer `$?` par erreur (en lançant quelque chose via `system`).

Notez que lorsque vous utilisez `-n` et `-p` avec Perl, `BEGIN` et `END` marchent exactement de la même façon qu'avec `awk`, sous forme dégénérée. De la façon dont sont réalisés (et sujet a changer, vu que cela ne pourrais être pire), les blocs `BEGIN` et `END` sont exécutés lorsque vous utilisez `-c` qui ne fait que tester la syntaxe, bien que votre code principal ne soit pas exécuté.

### 51.1.4 Classes Perl

Il n'y a pas de syntaxe de classe spéciale en Perl, mais un paquetage peut fonctionner comme une classe si il fournis des fonctions agissant comme des méthodes. Un tel paquetage peut dériver quelques unes de ses méthodes d'une autre classe (paquetage) en incluant le nom de l'autre paquetage dans son tableau global `@ISA` (qui doit être global, pas lexical).

Pour plus de détails, référez vous à *perltoot* et *perlobj*.

### 51.1.5 Modules Perl

Un module est juste un paquetage qui est défini dans un fichier de même nom, et qui est destiné à être réutilisé. Il peut arriver a cette effet en fournissant un mécanisme qui exportera certains de ses symboles dans la table de symboles du paquetage qui l'utilise. Ou bien, il peut fonctionner comme une classe et rendre possible l'accès a ses variables via des appels de fonctions, sans qu'il soit nécessaire d'exporter un seul symbole. Il peut bien sur faire un peu des deux.

Par exemple, pour commencer un module normal appelé `Some::Module`, Créez un fichier appelé `Some/Module.pm` et commencez avec ce patron :

```

package Some::Module; # suppose Some/Module.pm

use strict;

BEGIN {
 use Exporter ();
 use vars qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

```

```

On définit une version pour les vérifications
$VERSION = 1.00;
Si vous utilisez RCS/CVS, ceci serait préférable
le tout sur une seule ligne, pour MakeMaker
$VERSION = do { my @r = (q$Revisio: XXX $ =~ /\d+/g); sprintf "%d"."%02d" x $#r, @r };

@ISA = qw(Exporter);
@EXPORT = qw(&func1 &func2 &func4);
%EXPORT_TAGS = (); # ex. : TAG => [qw!name1 name2!],

vos variables globales à être exportées vont ici,
ainsi que vos fonctions, si nécessaire
@EXPORT_OK = qw($Var1 %Hashit &func3);
}
use vars @EXPORT_OK;

Les globales non exportées iront là
use vars qw(@more $stuff);

Initialisation de globales, en premier, celles qui seront exportées
$Var1 = '';
%Hashit = ();

Ensuite, les autres (qui seront accessibles via $Some::Module::stuff)
$stuff = '';
@more = ();

Toutes les lexicales doivent être créées avant
les fonctions qui les utilisent.

les lexicales privées vont là
my $priv_var = '';
my %secret_hash = ();

Voici pour finir une fonction interne à ce fichier,
Appelée par &$priv_func; elle ne peut être prototypée.
my $priv_func = sub {
 # des trucs ici.
};

faites toutes vos fonctions, exportées ou non;
n'oubliez pas de mettre quelque chose entre les {}
sub func1 {} # pas de prototype
sub func2() {} # proto void
sub func3($$) {} # proto avec 2 scalaires

celle là n'est pas exportée, mais peut être appelée !
sub func4(\%) {} # proto'd avec 1 hash par référence

END { } # on met tout pour faire le ménage ici (destructeurs globaux)

```

Enfin, continuez en déclarant et en utilisant vos variables dans des fonctions sans autres qualifications. Référez-vous à *Exporter* et *perlmodlib* pour plus de détails sur les mécanismes et les règles de style à adopter lors de la création de modules.

Les modules Perl sont inclus dans vos programmes en disant

```
use Module;
```

ou

```
use Module LIST;
```

Ce qui revient exactement à dire

```
BEGIN { require Module; import Module; }
```

ou

```
BEGIN { require Module; import Module LIST; }
```

Et plus spécifiquement

```
use Module ();
```

est équivalent à dire

```
BEGIN { require Module; }
```

Tous les modules perl ont l'extension *.pm*. `use` le suppose pour que vous n'ayez pas à taper "*Module.pm*" entre des guillemets. Ceci permet aussi de faire la différence entre les nouveaux modules des vieux fichiers *.pl* et *.ph*. Les noms de modules commencent par une majuscule, à moins qu'ils fonctionnent comme pragmas, les "Pragmas" sont en effet des directives du compilateur, et sont parfois appelés "modules pragmatiques" (ou même "pragmata" si vous êtes puristes).

Les deux déclarations :

```
require UnModule;
require "UnModule.pm";
```

diffèrent en deux points. Dans le premier cas, les doubles deux points dans le nom du module, comme dans `Un::Module`, sont transformés en séparateur système, généralement `/`. Le second ne le fait pas, ce devra être fait manuellement. La deuxième différence est que l'apparition du premier `require` indique au compilateur que les utilisations de la notation objet indirecte impliquant "UnModule", comme dans `$ob = purge UnModule`, sont des appels de méthodes et non des appels de fonctions. (Oui, ceci peut vraiment faire une différence).

Parce que l'instruction `use` implique un bloc `BEGIN`, l'importation des sémantiques intervient au moment où le `use` est compilé. C'est de cette façon qu'il lui est possible de fonctionner comme pragma, et aussi la manière dont laquelle les modules sont capables déclarer des fonctions qui seront visibles comme des opérateurs de liste pour le reste du fichier courant. Ceci ne sera pas vrai si vous utilisez `require` à la place de `use`. Avec `require`, vous allez au devant de ce problème :

```
require Cwd; # rends Cwd:: accessible
$here = Cwd::getcwd();

use Cwd; # importe les noms depuis Cwd::
$here = getcwd();

require Cwd; # rends Cwd:: accessible
$here = getcwd(); # oups ! y'a pas de main::getcwd()
```

En général, `use Module ()` est recommandé à la place de `require Module`, car cela détermine si le module est là au moment de la compilation, pas en plein milieu de l'exécution de votre programme. Comme exception, je verrais bien, le cas où deux modules essaient de se `use` l'un l'autre, et que chacun appelle une fonction de l'autre module. Dans ce cas, il est facile d'utiliser `require` à la place.

Les paquetages Perl peuvent être inclus dans d'autres paquetages, on peut donc avoir des noms de paquetages contenant `::` mais si nous utilisons le nom du paquetage directement comme nom de fichier, cela donnera des noms peu manipulables, voir impossibles sur certains systèmes. Par conséquent, si le nom d'un module est, disons, `Texte::Couleur`, alors, la définition se trouvera dans le fichier *Texte/Couleur.pm*.

Les modules Perl ont toujours un fichier *.pm*, mais ils peuvent aussi être des exécutables dynamiquement liés, ou des fonctions chargées automatiquement associées au module. Si tel est le cas, ce sera totalement transparent pour l'utilisateur du module. c'est le fichier *.pm* qui doit se charger de charger ce dont il a besoin. Le module `POSIX` est en fait, dynamique et autochargé, mais l'utilisateur a juste à dire `use POSIX` pour l'avoir.

Pour plus d'informations sur l'écriture d'extensions, référez vous à *perlxsut* et *perlguts*.

## 51.2 VOIR AUSSI

*perlmodlib* pour les questions générales sur comment faire des modules et des classes Perl, ainsi que la description de la librairie standard et du CPAN, *Exporter* pour savoir comment marche le mécanisme d'import/export de Perl, *perltoot* pour des explications en profondeur sur comment créer des classes, *perlobj* pour un document de référence sur les objets, et *perlsub* pour une explication sur les fonction et la portée de celle ci.

## 51.3 TRADUCTION

### 51.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 51.3.2 Traducteur

Mathieu Arnold <arn\_mat@club-internet.fr>

### 51.3.3 Relecture

Simon Washbrook <swashbro@tlse.marben.fr>



# Chapitre 52

## perlmodlib

Pour construire de nouveaux modules et trouver les existants

### 52.1 DESCRIPTION

### 52.2 LA LIBRAIRIE DE MODULES PERL

Un certain nombre de modules sont inclus dans la distribution de Perl. Ils sont décrit plus loin, à la fin du *.pm*. Vous pouvez alors découvrir les fichiers dans le répertoire des bibliothèques qui se terminent par autre chose que *.pl* ou *.ph*. Ce sont d'anciennes bibliothèques fournies pour que les anciens programmes les utilisant continuent de fonctionner. Les fichiers *.pl* ont tous été converti en modules standards, et les fichiers *.ph* fabriqués par **h2ph** sont probablement terminés comme extension des modules fabriqués par **h2xs**. (Certains *.ph* peuvent être déjà disponibles par le module POSIX. Le fichier **pl2pm** de la distribution peut vous aider dans votre conversion, mais il s'agit que d'un mécanisme du processus et par conséquent loin d'être une preuve infaillible.

#### 52.2.1 Pragmatic Modules

Ils fonctionnent comme les pragmas par le fait qu'ils ont tendance à affecter la compilation de votre programme, et ainsi fonctionnent habituellement bien que lorsqu'ils sont utilisés avec `use`, ou `no`. La plupart de ceux-ci ont une portée locale, donc un BLOCK interne peut outrepasser n'importe lequel en faisant:

```
no integer;
no strict 'refs';
```

ce qui dure jusqu'à la fin de ce BLOC.

À la différence des pragmas qui effectuent \$^H variable de conseils, les déclarations `use vars` et `use subs` ne sont pas limitées au bloc. Elles vous permettent de prédéclarer des variables ou des sous-programmes dans un *fichier* plutôt que juste pour un bloc. De telles déclarations sont pertinentes pour le fichier entier dans lequel elles ont été déclarées. Vous ne pouvez pas les annuler avec `no vars` ou `no subs`.

Les pragmas suivants sont définis (et ont leur propre documentation).

#### **use autouse MODULE => qw(sub1 sub2 sub3)**

Reporte `require MODULE` jusqu'à ce que quelqu'un appelle un des sous-programmes indiqués (qui doivent être exportés par MODULE). Ce pragma devrait être utilisé avec prudence, et seulement si nécessaire.

#### **blib**

manipule @INC au moment de la compilation pour utiliser la version déinstallée par MakeMaker's d'une paquetage.

#### **diagnostics**

force le diagnostic explicite des messages d'alerte

#### **integer**

Calcul arithmétique en integer au lieu de double

#### **less**

demande moins de quelque chose au compilateur

**lib**

manipule @INC au moment de la compilation

**locale**

utilisez ou ignorez l'état actuel de locale pour des opérations internes (voir *perllocale*)

**ops**

limitez les opcodes nommés quand vous compilez ou exécutez du code Perl

**overload**

surcharge les opérations basic de Perl

**re**

modifie le comportement des expression rationnelles

**sigtrap**

permet la capture de signaux simples

**strict**

restreint les constructions non sûres

**subs**

prédéclare les noms de fonctions

**vmsish**

adopte certains comportements spécifiques à VMS

**vars**

prédéclare les noms de variables globales

### 52.2.2 Standard Modules

En standard, on s'attend à ce que des modules empaquetés se comportent tous d'une façon bien définie en ce qui concerne la pollution de namespace parce qu'ils utilisent le module Exporter. Voir leur propre documentation pour des détails.

**AnyDBM\_File**

fournissez le cadre de travail pour de multiples DBMs

**AutoLoader**

charge les fonctions seulement à la demande

**AutoSplit**

scinde un paquetage pour le chargement automatique

**Benchmark**

benchmark pour tester les temps d'exécution de votre code

**CPAN**

interface pour le Comprehensive Perl Archive Network

**CPAN::FirstTime**

crée un fichier de configuration CPAN

**CPAN::Nox**

exécutez CPAN tout en évitant les extensions compilées

**Carp**

prévient les erreurs (de la perspective de l'appelant)

**Class::Struct**

déclare des types de données similaires au struct

**Config**

accède aux informations de configuration de Perl

**Cwd**

donne le nom du répertoire de travail courant

**DB\_File**

accède à Berkeley DB

**Devel::SelfStubber**

génère les stubs pour un module qui se charge lui-même (SelfLoading)

**DirHandle**

fournit les méthodes de objets pour les descripteurs de répertoires

**DynaLoader**

charge dynamiquement les librairies C dans le code Perl

**English**

utilise les noms anglais (ou awk) jolies pour des variables laides de ponctuation

**Env**

importe les variables d'environnement

**Exporter**

implémente la méthode d'import par défaut des modules

**ExtUtils::Embed**

utilitaires pour encapsuler du Perl dans les applications C/C++

**ExtUtils::Install**

installez des fichiers d'ici à là

**ExtUtils::Liblist**

détermine les librairies à utiliser et comment les utiliser

**ExtUtils::MM\_OS2**

méthode pour écraser le comportement d'Unix dans ExtUtils::MakeMaker

**ExtUtils::MM\_Unix**

méthodes utilisées par ExtUtils::MakeMaker

**ExtUtils::MM\_VMS**

méthode pour écraser le comportement d'Unix dans ExtUtils::MakeMaker

**ExtUtils::MakeMaker**

crée une extension de Makefile

**ExtUtils::Manifest**

utilitaires pour écrire et vérifier un fichier MANIFEST

**ExtUtils::Mkbootstrap**

fabrique un fichier d'amorçage à l'usage de DynaLoader

**ExtUtils::Mksymlists**

écrivez les fichiers d'options d'éditeur de liens pour les extensions dynamiques

**ExtUtils::testlib**

ajoute les répertoire blib/\* à @INC

**Fatal**

Transforme les erreurs dans les fonctions internes ou dans les fonctions de Perl fatales

**Fcntl**

Charge les définitions de C Fcntl.h

**File::Basename**

sépare un nom de répertoire en parties

**File::CheckTree**

effectue plusieurs contrôles sur des tests de fichiers dans un arbre

**File::Compare**

compare des fichiers ou des descripteurs de fichiers

**File::Copy**

copie des fichiers ou des descripteurs de fichiers

**File::Find**

traverse un arbre de fichiers

**File::Path**

crée ou supprime une série de répertoires

**File::stat**

by-name interface to Perl's builtin stat() functions

**FileCache**

maintenez plus de fichiers ouverts que les autorisations du système le permettent

**FileHandle**

fournit les méthodes des objets pour les descripteurs de fichiers

**FindBin**

localise le répertoire original du script Perl

**GDBM\_File**

accède à la librairie gdbm

**Getopt::Long**

traitement étendu des options de ligne de commande

**Getopt::Std**

commutateurs (switches) de processus de caractères simples avec groupe de commutateurs (switch clustering)

**I18N::Collate**

comparez des données scalaires de 8 bits selon la configuration locale actuelle

**IO**

charge divers modules d'E/S

**IO::File**

fournit les méthodes d'objets pour les descripteurs de fichiers

**IO::Handle**

fournit les méthodes des objets pour les opérations d'E/S

**IO::Pipe**

fournit les méthodes des objets pour les tubes (pipe)

**IO::Seekable**

fournit les méthodes pour les objets d'E/S

**IO::Select**

interface OO pour l'appel système sélectionné

**IO::Socket**

interface des objets pour les communications par socket

**IPC::Open2**

ouvre un process pour à la fois lire et écrire

**IPC::Open3**

ouvre un process pour lire, écrire et capturer les erreurs

**Math::BigFloat**

module pour les nombres à virgule de longueur arbitraire

**Math::BigInt**

module pour les entiers de taille arbitraire

**Math::Complex**

module pour les nombres complexes et les fonctions mathématiques associées

**Math::Trig**

interface simple pour Math::Complex pour ceux qui ont besoin des fonctions trigonométriques seulement pour les nombres réels

**NDBM\_File**

lie l'accès aux fichier ndbm

**Net::Ping**

Bonjour, il y a quelqu'un ?

**Net::hostent**

interface par nom pour les fonctions internes de Perl gethost\*()

**Net::netent**

interface par nom pour les fonctions internes de Perl getnet\*()

**Net::protoent**

interface par nom pour les fonctions internes de Perl getproto\*()

**Net::servent**

interface par nom pour les fonctions internes de Perl getserv\*()

**Opcode**

désactive les opcodes nommés pendant la compilation ou l'exécution de code Perl

**Pod::Text**

converti des données POD en texte ASCII formaté

**POSIX**

interface pour le standard IEEE 1003.1

**SDBM\_File**

lie l'accès au fichiers sdbms

**Safe**

compile et exécute le code dans des compartiments restreints

**Search::Dict**

cherche une clef dans le fichier du dictionnaire

**SelectSaver**

sauve et restaure le descripteur de fichier sélectionné

**SelfLoader**

charge les fonctions seulement à la demande

**Shell**

lance des commandes shell de façon transparente dans Perl

**Socket**

charge la définition et les manipulateurs de structure de socket.h

**Symbol**

manipule les symboles Perl et leurs noms

**Sys::Hostname**

essaye toutes les méthodes conventionnelles pour obtenir un nom de machine

**Sys::Syslog**

interface pour les appels à la commande Unix syslog(3)

**Term::Cap**

interface pour termcap

**Term::Complete**

module de complétion de mots

**Term::ReadLine**

interface vers des paquetages readline variés

**Test::Harness**

Lance des scripts de tests standards de Perl avec des statistiques

**Text::Abbrev**

créé une table d'abréviation d'une liste

**Text::ParseWords**

parse du texte dans un tableau de marques

**Text::Soundex**

implémentation de l'algorithme Soundex Algorithm comme décrit par Knuth

**Text::Tabs**

agrandit ou diminue des tableaux avec la fonction Unix expand(1) et unexpand(1)

**Text::Wrap**

formate les lignes pour former des paragraphes simples

**Tie::Hash**

définitions de base des classes pour les tableaux associatifs liés (tied hashes)

**Tie::RefHash**

définitions de base des classes pour définitions de base des classes pour les tableaux associatifs liés (tied hashes) avec comme références les clés avec des références comme clés

**Tie::Scalar**

définitions de base des classes pour scalaires liés (tied)

**Tie::SubstrHash**

tableau associatif avec taille-de-tableau-fixe, longueur-de-clé-fixe

**Time::Local**

calcul efficace de l'heure locale et GMT

**Time::gmtime**

interface par nom pour les fonctions internes de Perl gmtime()

**Time::localtime**

interface par nom pour les fonctions internes de Perl localtime()

**Time::tm**

objet interne utilisé par Time::gmtime et Time::localtime

**UNIVERSAL**

classe de base pour TOUTES les classes (références bénites (blessed))

**User::grent**

interface par nom pour les fonctions internes de Perl getgr\*()

**User::pwent**

interface par nom pour les fonctions internes de Perl getpw\*()

Pour trouver *tous* les modules installés sur votre système, incluant ceux sans documentation ou en dehors de la release standard, faites ceci:

```
% find 'perl -e 'print "@INC"' -name '*.pm' -print
```

Ils doivent avoir leur propre documentation installée et accessible via votre commande système man(1). Si cela échoue, essayer le programme *perldoc*.

### 52.2.3 Extension de Modules

Les extensions de modules sont écrits en C (ou un mixte de Perl et de C) et peuvent être liées (linked) statiquement ou en général sont chargées dynamiquement dans Perl si et quand vous en avez besoin. Les extensions de modules supportées comprennent les Socket, Fcntl, et les modules POSIX.

La plupart des extensions C de modules populaires n'arrivent pas tout prêt (ou du moins, pas complètement) due à leur taille, leur volatilité, ou simplement par manque de temps de tests adéquats et de configuration autour des multitudes de plates-formes où Perl est beta-testé. Vous êtes encouragé à les regarder dans archie(1L), la FAQ Perl ou Meta-FAQ, les pages WWW, et même avec leur auteurs avant de poster des questions pour leurs conditions et dispositions actuelles.

## 52.3 CPAN

CPAN signifie le Comprehensive Perl Archive Network. Il s'agit d'une réplique globale de tous les matériaux Perl connus, incluant des centaines de modules non chargés. Voici les catégories principales de ces modules:

- Les Extensions de langage et la Documentation des outils
- Support au Développement
- Interface pour le Système d'exploitation
- Réseau, contrôle de modems and Processus d'intercommunication
- Types de données et utilitaires de type de données
- Interfaces base de données
- User Interfaces
- Interfaces pour / Emulations d'autres langages de programmation
- Nom de fichiers, Système de fichiers et verrous de fichiers (voir aussi Descripteur de fichiers)
- Traitements de chaînes de caractères, traitements de textes de langage, analyse, et recherche
- Option, argument, paramètre, et traitement de fichier de configuration
- Internationalisation et Locale
- Authentification, Sécurité, and Encryption
- World Wide Web, HTML, HTTP, CGI, MIME

- Serveur and utilitaires de Démons
- Archivage et Compression
- Images, Manipulation de Pixmap et Bitmap, Dessins et Graphiques
- Mail et News Usenet
- Utilitaires de Contrôle de Flux (callbacks et exceptions etc)
- Utilitaires pour les descripteurs de fichier ou pour les chaînes d'entrée/sortie
- Modules variés

Les sites officiels CPAN en date de cette écriture sont les suivants. Vous devriez essayer de choisir un près de chez vous:

- Afrique
  - Afrique du Sud <ftp://ftp.is.co.za/programming/perl/CPAN/>
- Asie
  - Hong Kong <ftp://ftp.hkstar.com/pub/CPAN/>
  - Japon <ftp://ftp.jaist.ac.jp/pub/lang/perl/CPAN/>
  - <ftp://ftp.lab.kdd.co.jp/lang/perl/CPAN/>
  - Corée du Sud <ftp://ftp.nuri.net/pub/CPAN/>
  - Taiwan <ftp://dongpo.math.ncu.edu.tw/perl/CPAN/>
  - <ftp://ftp.wownet.net/pub2/PERL/>
- Australie
  - Australie <ftp://ftp.netinfo.com.au/pub/perl/CPAN/>
  - Nouvelle Zélande <ftp://ftp.tekotago.ac.nz/pub/perl/CPAN/>
- Europe
  - Autriche <ftp://ftp.tuwien.ac.at/pub/languages/perl/CPAN/>
  - Belgique <ftp://ftp.kulnet.kuleuven.ac.be/pub/mirror/CPAN/>
  - Rép. Tchèque <ftp://sunsite.mff.cuni.cz/Languages/Perl/CPAN/>
  - Danemark <ftp://sunsite.auc.dk/pub/languages/perl/CPAN/>
  - Finlande <ftp://ftp.funet.fi/pub/languages/perl/CPAN/>
  - France <ftp://ftp.ibp.fr/pub/perl/CPAN/>
  - <ftp://ftp.pasteur.fr/pub/computing/unix/perl/CPAN/>
  - Allemagne <ftp://ftp.gmd.de/packages/CPAN/>
  - <ftp://ftp.leo.org/pub/comp/programming/languages/perl/CPAN/>
  - <ftp://ftp.mpi-sb.mpg.de/pub/perl/CPAN/>
  - <ftp://ftp.rz.ruhr-uni-bochum.de/pub/CPAN/>
  - <ftp://ftp.uni-erlangen.de/pub/source/Perl/CPAN/>
  - <ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/>
  - Grèce <ftp://ftp.ntua.gr/pub/lang/perl/>
  - Hongrie <ftp://ftp.kfki.hu/pub/packages/perl/CPAN/>
  - Italie <ftp://cis.utoverm.it/CPAN/>
  - the Netherlands <ftp://ftp.cs.ruu.nl/pub/PERL/CPAN/>
  - <ftp://ftp.EU.net/packages/cpan/>
  - Norvège <ftp://ftp.uit.no/pub/languages/perl/cpan/>
  - Pologne <ftp://ftp.pk.edu.pl/pub/lang/perl/CPAN/>
  - <ftp://sunsite.icm.edu.pl/pub/CPAN/>
  - Portugal <ftp://ftp.ci.uminho.pt/pub/lang/perl/>
  - <ftp://ftp.telepac.pt/pub/CPAN/>
  - Russie <ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/>
  - Slovénie <ftp://ftp.arnes.si/software/perl/CPAN/>
  - Espagne <ftp://ftp.etse.urv.es/pub/mirror/perl/>
  - <ftp://ftp.rediris.es/mirror/CPAN/>
  - Suède <ftp://ftp.sunet.se/pub/lang/perl/CPAN/>
  - RU <ftp://ftp.demon.co.uk/pub/mirrors/perl/CPAN/>
  - <ftp://sunsite.doc.ic.ac.uk/packages/CPAN/>
  - <ftp://unix.hensa.ac.uk/mirrors/perl-CPAN/>
- Amérique du Nord
  - Ontario <ftp://ftp.utilis.com/public/CPAN/>
  - <ftp://enterprise.ic.gc.ca/pub/perl/CPAN/>
  - Manitoba <ftp://theory.uwinnipeg.ca/pub/CPAN/>
  - Californie <ftp://ftp.digital.com/pub/plan/perl/CPAN/>
  - <ftp://ftp.cdrom.com/pub/perl/CPAN/>
  - Colorado <ftp://ftp.cs.colorado.edu/pub/perl/CPAN/>
  - Floride <ftp://ftp.cis.ufl.edu/pub/perl/CPAN/>

Illinois	<code>ftp://uiarchive.uiuc.edu/pub/lang/perl/CPAN/</code>
Massachusetts	<code>ftp://ftp.iguide.com/pub/mirrors/packages/perl/CPAN/</code>
New York	<code>ftp://ftp.rge.com/pub/languages/perl/</code>
Caroline du Nord	<code>ftp://ftp.duke.edu/pub/perl/</code>
Oklahoma	<code>ftp://ftp.ou.edu/mirrors/CPAN/</code>
Oregon	<code>http://www.perl.org/CPAN/</code> <code>ftp://ftp.orst.edu/pub/packages/CPAN/</code>
Pennsylvanie	<code>ftp://ftp.epix.net/pub/languages/perl/</code>
Texas	<code>ftp://ftp.sedl.org/pub/mirrors/CPAN/</code> <code>ftp://ftp.metronet.com/pub/perl/</code>

– Amérique du Sud

Chili	<code>ftp://sunsite.dcc.uchile.cl/pub/Lang/perl/CPAN/</code>
-------	--------------------------------------------------------------

Pour une liste à jour des sites CPAN, voir <http://www.perl.com/perl/CPAN> ou <ftp://ftp.perl.com/perl/>.

## 52.4 Modules: Création, Utilisation, et Abus

(la section suivante est empruntée directement des fichiers des modules de Tim Buncees, disponible depuis votre site CPAN plus proche.)

Le Perl implémente une classe en utilisant un module, mais la présence d'un module n'implique pas la présence d'une classe. Un module est juste un espace de nom (namespace). Une classe est un module qui fournit les sous-programmes qui peuvent être utilisés comme méthodes. Une méthode est juste un sous-programme qui prévoit que son premier argument est le nom d'un module (pour des méthodes "statiques"), ou une référence à quelque chose (pour des méthodes "virtuelles").

Un module est un fichier qui (par convention) fournit une classe du même nom (sans le .pm), plus une méthode d'importation dans cette classe qui peut s'appeler pour chercher les symboles exportés. Ce module peut appliquer certaines de ses méthodes en chargeant les objets dynamiques en C ou en C++, mais cela devrait être totalement transparent à l'utilisateur du module. De même, le module pourrait installer une fonction AUTOLOAD dans des définitions de sous-programme à la demande, mais c'est également transparent. Seulement un fichier .pm est nécessaire pour exister. Voir *perlsub*, *perltoot*, et *AutoLoader* pour des détails au sujet du mécanisme de AUTOLOAD.

### 52.4.1 Directives pour la création de modules

#### Des modules similaires existent-ils déjà sous une certaine forme?

Si oui essayez, s'il vous plaît de réutiliser les modules existants en entier ou en héritant des dispositifs utiles dans une nouvelle classe. Si ce n'est pratique, voyez avec les auteurs de ce module pour travailler à étendre ou à mettre en valeur les fonctionnalités des modules existants. Un exemple parfait est la pléthore de modules dans perl4 pour traiter des options de ligne de commande.

Si vous écrivez un module pour étendre un ensemble de modules déjà existant, coordonnez-vous s'il vous plaît avec l'auteur du module. Cela aide si vous suivez la même convention de nom et d'interaction de module que l'auteur initial.

#### Essayez de concevoir le nouveau module pour être facile étendre et réutiliser.

Utilisez les références sacrifiées (blessed). Utilisez deux arguments pour sacrifier le nom de classe donné comme premier paramètre du constructeur, ex. :

```
sub new {
 my $class = shift;
 return bless {}, $class;
}
```

ou même ceci si vous voudriez qu'il soit utilisé comme méthode statique ou virtuelle :

```
sub new {
 my $self = shift;
 my $class = ref($self) || $self;
 return bless {}, $class;
}
```



Un passage de tableau comme références permet ainsi plus de paramètres pouvant être ajoutés plus tard (et également plus rapide). Convertissez les fonctions en méthodes le cas échéant. Coupez les grandes méthodes en les plus petites plus flexibles. Héritez des méthodes d'autres modules si approprié.

Évitez les essais nommés de classe comme: `die "Invalid" unless ref $ref eq 'FOO'`. D'une façon générale vous pouvez effacer la partie `"eq 'FOO'"` sans que cela pose problème. Laissez les objets s'occuper d'eux! D'une façon générale, évitez les noms codés en dur de classe aussi loin que possible.

Évitez `$r->Class::func()` en utilisant `@ISA=qw(... Class ...)` et `$r->func()` fonctionnera (voir *perlbot* pour plus de détails).

Utilisez `autosplit` pour les fonctions peu utilisées ou nouvellement ajoutées pour que cela ne soit pas un fardeau pour les programmes qui ne les utilisent pas. Ajoutez les fonctions de test au module après le `__END__` en utilisant `AutoSplit` ou en disant:

```
eval join(' ', <main::DATA>) || die $@ unless caller();
```

Votre module passe-t-il le test 'de la sous classe vide? Si vous dites `"@SUBCLASS::ISA = qw(YOURCLASS);"` vos applications devraient pouvoir utiliser la SOUS-CLASSE exactement de la même façon que YOURCLASS. Par exemple, est-ce que votre application fonctionne toujours si vous changez: `$obj = new VOTRECLASSE;` en: `$obj = new SOUS-CLASSE;`

Évitez de maintenir n'importe quelle information d'état dans vos modules. Cela le rend difficile d'utilisation pour de multiples autres modules. Gardez à l'esprit l'information d'état de subsistance dans les objets.

Essayez toujours d'utiliser `-w`. Essayez d'utiliser `use strict;` (ou `use strict qw(...);`). Souvenez-vous que vous pouvez ajouter `no strict qw(...);` aux blocs individuels de code qui nécessitent moins de contraintes. Utilisez toujours `-w`. Utilisez toujours `-w!` Suivez les directives de `perlstyle(1)`.

### Quelques directives simples de modèle

Le manuel de `perlstyle` fourni avec Perl a beaucoup de points utiles.

La façon de coder est une question de goût personnel. Beaucoup de gens font évoluer leur style sur plusieurs années pendant qu'elles apprennent ce qui les aide à écrire et mettre à jour un bon code. Voici un ensemble de suggestions assorties qui semblent être largement répandues par les réalisateurs expérimentés:

Employez les underscores pour séparer des mots. Il est généralement plus facile de lire `$un_nom_de_variable` que `$UnNomDeVariable`, particulièrement pour les personnes de langue maternelle autre que l'anglais. C'est une règle simple qui fonctionne également avec `NOM_DE_VARIABLE`.

Les noms de Package/Module sont une exception à cette règle. Le Perl réserve officiellement des noms minuscules de module pour des modules de 'pragma' comme `number` entier et `strict`. D'autres modules normalement commencent par une majuscule et utilisent ensuite les cas mélangés sans des souligné (besoin d'être court et portable).

Vous pouvez trouver pratique d'utiliser la case des lettres pour indiquer la portée ou la nature d'une variable. Par exemple:

```
$TOUT_EN_MAJUSCULES : seulement les constantes (prenez garde aux
désaccords avec les variables de Perl)
$Seulement_Quelques_Majuscules portée le temps d'un paquetage,
variables globales/statiques
$aucune_majuscules portée d'une variable dans une fonction avec
my() ou local()
```

Les noms de fonction et de méthode semblent mieux fonctionner quand tout est en minuscule. ex., `$obj->as_string()`.

Vous pouvez employer un underscore devant le nom des variables pour indiquer qu'une variable ou une fonction ne devrait pas être utilisée en dehors du module qui l'a définie.

### Choisir quoi exporter.

N'exportez pas les noms de méthode!

N'exportez pas toute autre chose par défaut sans bonne raison!

Les exportations polluent le namespace de l'utilisateur du module. Si vous devez exporter quelque chose utiliser `@EXPORT_OK` de préférence à `@EXPORT` et éviter des noms communs ou courts pour réduire le risque de désaccords sur les noms.

D'une façon générale quelque chose non exporté est encore accessible de l'extérieur du module en utilisant la syntaxe de `ModuleName::item_name` (ou `$blessed_ref->method`). Par convention vous pouvez employer un underscore précédent le nom de variable pour indiquer officiellement qu'il s'agit de variables 'internes' et pas pour l'usage public.

(il est possible d'obtenir des fonctions privées en disant: `my $subref = sub { ... }; &$subref;`. Mais il n'y a aucune façon de les appeler directement comme méthode, parce qu'une méthode doit avoir un nom dans la table de symbole.)

En règle générale, si le module essaye d'être orienté objet alors n'exportez rien. S'il c'est juste une collection de fonctions alors @EXPORT\_OK quelque chose mais l'utilisation de @EXPORT est à faire avec prudence.

### Choisir un nom pour le module.

Ce nom devrait être descriptif, précis, et aussi complet que possible. Evitez n'importe quel risque d'ambiguïté. Essayez toujours d'utiliser deux ou plus de mots. D'une façon générale le nom devrait refléter ce qui est spécial au sujet de ce que le module fait plutôt que de la façon dont il le fait. Veuillez employer les noms emboîtés de module pour grouper officieusement ou pour classer un module par catégorie. Il devrait y a une très bonne raison pour un module de ne pas avoir un nom emboîté. Les noms de module devraient commencer par une majuscule.

Ayant 57 modules tous appelé Sort ne rendra pas la vie facile pour n'importe qui (avoir cependant 23 appelés Sort::Quick est seulement marginalement meilleur :-). Imaginez quelqu'un essayant d'installer votre module à côté de beaucoup d'autres. Si vous avez un doute demandez des suggestions dans comp.lang.perl.misc.

Si vous développez une suite de modules/classes liés, habituellement on utilise les classes emboîtées avec un préfixe commun car ceci évitera des désaccords de namespace. Par exemple: Xyz::Control, Xyz::View, Xyz::Model etc... Utilisez les modules dans cette liste comme guide nommant.

Si vous ajoutez un nouveau module à un ensemble, suivez les normes de l'auteur initial pour nommer les modules et l'interface des méthodes dans ces modules.

Pour être portable, chaque composant d'un nom de module devrait être limité à 11 caractères. S'il pourrait être employé sur MS-DOS alors vous devez vous assurer que chacun fait moins de 8 caractères. Les modules emboîtés facilitent ceci.

### Est-ce que vous avez bien fait ?

Comment savez-vous que vous avez pris les bonnes décisions? Avez-vous sélectionné une conception d'interface qui posera des problèmes plus tard? Avez-vous sélectionné le nom le plus approprié? Avez-vous des questions?

La meilleure façon de savoir est de prendre beaucoup de suggestions utiles, et de demander à quelqu'un qui sait. Comp.lang.perl.misc est lu par toutes les personnes qui développent des modules et c'est le meilleur endroit pour demander.

Tout que vous devez faire est de poster un court sommaire du module, de son but et de ses interfaces. Quelques lignes sur chacune des méthodes principales est probablement suffisant. (si vous signalez le module entier il pourrait être ignoré par les personnes occupées - généralement les mêmes personnes dont vous aimeriez avoir l'avis !)

Ne vous inquiétez dans votre post si vous ne pouvez pas dire quand le module sera prêt - juste dites-le dans le message. Il pourrait être intéressant d'inviter d'autres pour vous aider, ils peuvent le terminer pour vous!

### README et autres fichiers additionnels.

Il est bien connu que les développeurs de logiciels documentent habituellement entièrement le logiciel qu'ils écrivent. Si cependant le monde est dans le besoin pressant de votre logiciel et qu'il n'y a pas assez de temps pour écrire toute la documentation s'il vous plaît au moins fournissez un fichier README qui contient:

- Une description du module/paquetage/extension etc.
- Une note sur le copyright - voir plus loin.
- Prérequis - ce dont vous pouvez avoir besoin.
- Comment le construire - les changements éventuels dans Makefile.PL etc.
- Comment l'installer.
- Les changements récents de cette version, spécialement les incompatibilités.
- Changements / améliorations que vous prévoyez de faire dans le futur.

Si le fichier README semble devenir trop large, séparer le en plusieurs sections dans des fichiers séparés: INSTALL, Copying,(A copier) ToDo(A faire) etc.

### Ajouter une note sur le copyright.

Comment vous décidez du type de licence est une décision personnelle. Le mécanisme général est d'insérer votre Copyright et de faire une déclaration aux autres qu'ils peuvent copier/utiliser/modifier votre travail.

Perl, par exemple, est fournie avec deux types de licence: GNU GPL et The Artistic Licence (voir les fichiers README, Copying, et Artistic). Larry a de bonnes raisons pour ne pas utiliser que GNU GPL.

Ma recommandation personnelle, en dehors du respect pour Larry, est que Perl, et la communauté Perl au sens large est simplement défini comme tel:

```
Copyright (c) 1995 Your Name. All rights reserved.
This program is free software; you can redistribute it and/or
modify it under the same terms as Perl itself.
```

Ce texte devrait au moins apparaître dans le fichier README. Vous pouvez également souhaiter l'inclure dans un fichier Copying et dans vos fichiers sources. Rappelez-vous d'inclure les autres mots en plus de copyright.

**Donnez au module un nombre de version/issue/release.**

Pour être entièrement compatible avec les modules Exporter et MakeMaker vous devriez enregistrer le numéro de version de votre module dans une variable non-my appelée \$VERSION. Ceci devrait être un nombre à virgule flottante avec au moins deux chiffres après la décimale (c.-à-d., centième, par exemple, \$VERSION = " 0,01 " ). n'utilisez pas une version du modèle "1.3.2". Voir Exporter.pm dans Perl5.001m ou plus pour des détails.

Il peut être pratique pour ajouter une fonction ou méthode de rechercher le nombre. Utilisez le nombre dans les annonces et les noms de fichier d'archives quand vous faites une version d'un module (ModuleName-1.02.tar.Z). Voir perldoc ExtUtils::MakeMaker.pm pour des détails.

**Comment construire et distribuer un module.**

C'est une bonne idée de poster une annonce de la disponibilité de votre module (ou du module lui-même si il est petit) dans le groupe de discussion comp.lang.perl.announce. Ceci assurera au moins une très large distribution en dehors de la distribution Perl.

Si possible vous pouvez placer le module dans un des archives importantes ftp et inclure les détails de son emplacement dans votre annonce.

Quelques notes au sujet des archives ftp: Veuillez utiliser un nom de fichier descriptif qui inclut le numéro de version. La plupart des répertoires entrants ne seront pas lisibles/listables, c.-à-d., vous ne pourrez pas voir votre fichier après l'avoir téléchargé. Rappelez-vous d'envoyer votre message d'avis par mail aussitôt que possible après avoir téléchargé votre module, autrement votre fichier peut obtenir effacé automatiquement. Accordez du temps pour que le fichier soit traité et/ou contrôlez que le fichier a été traité avant d'annoncer son emplacement.

FTP Archives for Perl Modules:

Suivre les instructions et les liens sur

<http://franz.ww.tu-berlin.de/modulelist>

ou posez le dans un de ces sites:

<ftp://franz.ww.tu-berlin.de/incoming>  
<ftp://ftp.cis.ufl.edu/incoming>

et prévenez <[upload@franz.ww.tu-berlin.de](mailto:upload@franz.ww.tu-berlin.de)>.

En utilisant l'interface WWW vous pouvez demander au serveur de téléchargement de refléter vos modules de votre ftp ou de votre site Web dans votre propre répertoire sur CPAN!

Rappelez-vous s'il vous plaît de m'envoyer une entrée mise à jour pour la liste de modules!

**Faites attention quand vous faites une nouvelle version d'un module.**

Tâchez toujours de rester compatible avec les versions précédentes. Autrement essayez d'ajouter un mécanisme pour retourner à l'ancien comportement si les gens comptent là-dessus. Documentez les changements incompatibles.

**52.4.2 Directives pour convertir des bibliothèques Perl 4 en modules****Il n'y a pas de prérequis pour convertir quel module que ce soit.**

Si il n'est pas rompu, ne le fixez pas! Les bibliothèques Perl 4 devraient continuer à fonctionner sans problèmes. Vous pouvez avoir à faire quelques changements mineurs (comme changer les variables @ qui ne sont pas des tableaux en chaînes doublement cotées) mais il n'y a aucun besoin de convertir un fichier pl en module juste pour cela.

**Prendre en considération les implications.**

Toutes les applications de Perl qui se servent du script devront être changées (légèrement) si le script est converti en module. Cela vaut la peine si vous projetiez de faire d'autres changements en même temps.

**Tirez le meilleur de l'occasion.**

Si vous allez convertir un script en module vous pouvez profiter de l'occasion pour remodeler l'interface. Les 'directives pour la création de module' incluses plus haut plusieurs issues que vous devriez considérer.

**L'utilitaire pl2pm est votre point de départ.**

Cet utilitaire lira les fichiers \*.pl (donnés comme paramètres) et écrira les fichiers \*.pm correspondants. L'utilitaire pl2pm fait ce qui suit :

- Ajoute les lignes standards de prologue de module
- Convertissez les spécificateurs de package ' en ::
- Convertissez les die(...) en croak(...)

- Quelques autres changement mineurs

Le processus mécanique de pl2pm n'est pas une preuve garantie. Le code converti aura besoin de contrôles soigneux, particulièrement pour tous les rapports de module. N'effacez pas le fichier initial pl jusqu'à ce que le nouveau pm ne fonctionne!

### 52.4.3 Directives pour réutiliser le code d'application

- Des applications complètes sont rarement complètement sous la forme de bibliothèques/modules Perl.
- Beaucoup d'applications contiennent du code Perl qui pourrait être réutilisé.
- Aidez à sauver le monde! Partagez votre code sous une forme qui est facile à réutiliser.
- Débloquer le code réutilisable dans un ou plusieurs modules séparés.
- Saisissez l'occasion de reconsidérer et remodeler les interfaces.
- Dans certains cas 'l'application' peut alors être réduite à un petit fragment de code construits sur les modules réutilisables. Dans ce cas l'application pourrait être appelé comme :

```
% perl -e 'use Module::Name; method(@ARGV)' ...
```

ou

```
% perl -mModule::Name ... (perl5.002 ou plus récent)
```

## 52.5 NOTE

Le Perl n'impose pas de parties privées et publiques de ses modules comme vous avez pu voir dans d'autres langages comme C++, ADA, ou Modula-17. Le Perl n'a pas d'infatuation (satisfaction excessive et ridicule que l'on a de soi N.D.T) avec l'intimité imposée. Il préférerait que vous êtes restiez hors de sa salle à manger parce que vous n'avez pas été invités, pas parce qu'il a un fusil de chasse.

Le module et son utilisateur ont un contrat, dont une partie est un droit commun, et une partie qui est "écrite". Une partie du contrat de droit commun est qu'un module ne pollue aucun namespace qu'on ne lui ai pas demandé. Le contrat écrit pour le module (cad la documentation) peut prendre d'autres dispositions. Mais vous savez quand vous utilisez <use RedefineTheWorld>, vous redéfinissez le monde et voulez bien en accepter les conséquences.

## 52.6 TRADUCTION

### 52.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 52.6.2 Traducteur

Yves Maniette <yves@giga.sct.ub.es>

### 52.6.3 Relecture

Personne pour l'instant.

# Chapitre 53

## perlmodinstall

Installation des modules CPAN

### 53.1 DESCRIPTION

Un module peut être vu comme une unité de base de code Perl réutilisable ; voyez *perlmod* pour plus de détails. Dès lors que quelqu'un crée un ensemble de code Perl dont il pense que la communauté pourra y trouver une utilité, il s'inscrit comme développeur Perl à l'adresse <http://www.perl.com/CPAN/modules/04pause.html> de sorte qu'il puisse exporter ses lignes de code vers le CPAN. LE CPAN est le «Réseau Complet d'Archives Perl» (Comprehensive Perl Archive Network) qui peut être consulté à l'adresse <http://www.perl.com/CPAN/>. (Ce sigle ne sera pas traduit dans ce document, puisqu'il concerne bien souvent des URL)

La présente documentation a été écrite à l'intention de quiconque désire rapatrier sur sa machine des modules du CPAN pour les y installer.

#### 53.1.1 PRÉAMBULE

Vous venez donc de rapatrier un fichier dont l'extension est `.tar.gz` (ou parfois `.zip`) dont vous savez qu'il contient un beau module avec de vrais morceaux de code... Il vous faudra franchir les quatre étapes suivantes pour l'installer :

**DÉCOMPRIMER le fichier**

**DÉBALLER le fichier dans un répertoire**

**CONSTRUIRE le module (parfois ce n'est pas nécessaire)**

**INSTALLER le module.**

Voici comment faire chacune de ces étapes pour divers systèmes d'exploitation. Attention, ce texte *ne vous dispense pas* de lire les fichiers `README` et `INSTALL` qui pourraient être livrés avec le module, dans lesquels peuvent se trouver des informations plus spécifiques au module considéré.

Sachez aussi que ces instructions ont été prévues pour une installation dans le catalogue de modules Perl de votre système, mais vous pouvez installer des modules dans n'importe quel répertoire. Par exemple s'il est écrit ici `perl Makefile.PL`, vous pouvez remplacer cette commande par `perl Makefile.PL PREFIX=/mon/répertoire_perl`, ce qui aura pour effet l'installation des modules dans le répertoire `/mon/répertoire_perl`. Vous pourrez alors utiliser les modules à partir de vos programmes Perl avec la commande `use lib"/mon/répertoire_perl/lib/site_perl"`; or parfois encore plus simplement `use"/mon/répertoire_perl"`;

– **Si vous êtes sur Unix,**

Vous pouvez utiliser le module CPAN de Andreas Koenig's (<http://www.perl.com/CPAN/modules/by-module/CPAN>) pour exécuter automatiquement les étapes suivantes, de DÉCOMPRIMER à INSTALLER.

A. DÉCOMPRIMER

Decomprimez le fichier avec la commande `gzip -d votremodule.tar.gz`

On peut obtenir `gzip` à l'adresse <ftp://prep.ai.mit.edu/pub/gnu>.

Mais vous pouvez aussi combiner cette étape avec la suivante pour économiser de l'espace sur votre disque :

```
gzip -dc votremodule.tar.gz | tar -xof -
```

**B. DÉBALLER le module**

Déballez le fichier résultat avec la commande `tar -xof votremodule.tar`

**C. CONSTRUIRE le module**

Allez dans le répertoire nouvellement créé et écrivez :

```
perl Makefile.PL
make
make test
```

**D. INSTALLER le module**

Vous êtes encore dans le répertoire nouvellement créé ; exécutez alors :

```
make install
```

Assurez-vous que vous avez les permissions requises pour pouvoir installer le module dans le répertoire correspondant à la bibliothèque Perl 5. Bien souvent, il vous faudra être root.

Voilà tout ce que vous avez à faire sur un système Unix avec liens dynamiques. La plupart des systèmes Unix possèdent les liens dynamiques — si ce n'est pas le cas du vôtre, ou si pour une quelconque raison votre perl est à liens statiques et si le module nécessite une compilation, vous devrez construire un nouveau binaire Perl incluant le module. Là encore, vous devrez être root.

**– Si vous utilisez Windows 95 ou NT avec le portage ActiveState de Perl****A. DÉCOMPRIMER le module**

Vous pouvez utiliser Winzip (<http://www.winzip.com>) pour décompresser et déballer les modules.

**B. DÉBALLER le module**

WinZip l'aura déjà fait !

**C. CONSTRUIRE le module**

Le module doit-il être compilé ? (contient-il des fichiers avec les extensions suivantes : .xs, .c, .h, .y, .cc, .cxx, or .C ?) Si c'est le cas, c'est à vous de jouer ! Vous pouvez essayer de compiler le module vous-même si vous avez un compilateur de C, et en cas de succès, envoyer le code binaire résultat au CPAN, pour que d'autres puissent aussi l'utiliser. Si il n'y a pas besoin de compiler, vous n'avez rien à faire pour cette étape.

**D. INSTALLER le module**

Copiez le module dans le répertoire *lib* de Perl. C'est l'un des répertoires que vous voyez en exécutant la commande :

```
perl -e 'print "@INC"'
```

**– Si vous utilisez Windows 95 ou NT avec la distribution de base de Perl****A. DÉCOMPRIMER le module**

Au moment de rapatrier le fichier, assurez-vous qu'il termine bien avec l'extension `.tar.gz` ou `.zip`. Parfois, Windows enregistre les fichiers `.tar.gz` sous la forme `_tar.gz`, car les premières versions de Windows n'acceptaient pas plus d'un point dans les noms de fichiers.

Vous pouvez utiliser Winzip (<http://www.winzip.com>) pour décompresser et déballer les modules, mais vous pouvez aussi utiliser le logiciel unzip de Info-Zip (disponible à l'adresse <http://www.cdrom.com/pub/infozip/Info-Zip.html>) pour comprimer les fichiers `.zip` ; exécutez la commande `unzip votremodule.zip` dans l'interpréteur (shell).

Ou encore, si vous disposez de `tar` and `gzip`, vous pouvez exécuter :

```
gzip -cd votremodule.tar.gz | tar xvf -
```

dans l'interpréteur afin de décompresser `votremodule.tar.gz`. Cela aura pour effet de DÉBALLER le module par la même occasion.

**B. DÉBALLER le module**

Quelle que soit la méthode employée pour DÉCOMPRIMER le fichier résultat aura aussi été DÉBALLÉ dans la foulée sans que vous ayez besoin de faire quoi que ce soit.

**C. CONSTRUIRE le module**

Allez dans le répertoire nouvellement créé et lancez la commande :

```
perl Makefile.PL
dmake
dmake test
```

Selon la configuration de votre Perl, `dmake` peut ne pas être disponible. Vous devrez alors sans doute lui substituer ce que `perl -V:make` vous dira. En général, ce sera `nmake` ou `make`.

**D. INSTALLER le module**

Toujours dans ce répertoire, entrez :

```
dmake install
```

**– Si vous utilisez un Macintosh,****A. DÉCOMPRIMER le module**

Vous pouvez employer au choix StuffIt Expander (<http://www.aladdinsys.com/>) en combinaison avec *DropStuff with Expander Enhancer*, ou MacGzip (<http://persephone.cps.unizar.es/general/gente/spd/gzip/gzip.html>).

**B. DÉBALLER le module**

Si vous utilisez DropStuff ou Stuffit, il vous suffit d'extraire l'archive `.tar`, mais vous pouvez aussi utiliser *suntar* (<http://www.cirfid.unibo.it/~speranza>).

## C. CONSTRUIRE le module

## 1. Si Le module a besoin d'être compilé

Idée de base: vous aurez besoin de MPW et d'une combinaison de compilateurs CodeWarrior récent et ancien pour MPW et ses bibliothèques. Les Makefiles créés pour construire sous MPW utilisent les compilateurs Metrowerk. Il est vraisemblablement possible de compiler sans autres compilateurs, mais cela n'a pas encore été fait avec succès, à notre connaissance. Voyez la documentation disponible dans "MacPerl: Power and Ease" (à l'adresse <http://www.ptf.com/macperl/>) au sujet des extensions de portage et compilation, ou cherchez une bibliothèque déjà compilée, ou trouvez quelqu'un qui le compile pour vous.

Ou encore, demandez aux abonnés à la liste mac-perl ([mac-perl@iis.ee.ethz.ch](mailto:mac-perl@iis.ee.ethz.ch)) de le compiler pour vous. Pour s'abonner à cette liste, écrivez à [mac-perl-request@iis.ee.ethz.ch](mailto:mac-perl-request@iis.ee.ethz.ch).

## 2. Si le module n'a pas besoin d'être compilé, continuez ci-dessous.

## D. INSTALLER le module

Assurez-vous que les caractères de retour-chariot sont bien en format Mac et non pas en format Unix. Déplacez manuellement les fichiers dans les dossiers correspondants.

Déplacez les fichiers vers leur destination finale. Il s'agira probablement de `$ENV{MACPERL}site_lib:` (c'est à dire `HD:MacPerl site:site_lib:`). Vous pouvez ajouter de nouveaux chemins à ceux par défaut `@INC` dans le menu des préférences de l'application MacPerl (`$ENV{MACPERL}site_lib:` est ajouté automatiquement). Créer toute structure de répertoire requise (par exemple, pour `Some::Module`, créez `$ENV{MACPERL}site_lib:Some:` et placez `Module.pm` dans ce répertoire).

Exécutez le script suivant (ou quelque chose de semblable):

```
#!/perl -w
use AutoSplit;
my $dir = "${MACPERL}site_perl";
autosplit("${dir}:Some:Module.pm", "${dir}:auto", 0, 1, 1);
```

Un beau jour il finira bien par y avoir une façon d'automatiser ce processus d'installation ; quelques solutions existent déjà, mais aucune d'entre-elles n'est encore prête pour le public.

## – Si vous êtes sur le portage DJGPP du DOS,

## A. DECOMPRIMER

Le programme `djtarx` (<ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/>) décompressera et déballera le module.

## B. DÉBALLER

Voir le paragraphe ci-dessus

## C. COMPILER

Allez dans le répertoire nouvellement créé et s<exécutez :>

```
perl Makefile.PL
make
make test
```

Vous aurez besoin des ensembles de programmes mentionnés dans `Readme.dos` dans la distribution Perl.

## D. INSTALLER

Toujours dans ce répertoire nouvellement créé, exécutez :

```
make install
```

Vous aurez besoin des ensembles de programmes mentionnés dans `Readme.dos` dans la distribution Perl.

## – Si vous êtes sur OS/2,

Procurez-vous la suite EMX de développement et `gzip/tar`, disponibles chez Hobbes (<http://hobbes.nmsu.edu>) ou Leo (<http://www.leo.org>), puis suivez les instructions pour Unix.

## – Si vous êtes sur VMS,

Au moment de rapatrier le fichier du CPAN, sauvegardez-le avec l'extension `.tgz` au lieu de `.tar.gz`. Tous les autres points dans le noms du fichier devront être remplacés par des tirets bas. Par exemple, le fichier `Votre-Module-1.33.tar.gz` devra être sauvegardé sous le nom `Votre-Module-1_33.tgz`.

## A. DECOMPRIMER

Exécutez :

```
gzip -d Votre-Module.tgz
```

ou, pour les modules zippés, exécutez :

```
unzip Votre-Module.zip
```

Les exécutable de `gzip`, `zip`, and `VMStar` (Alphas: <http://www.openvms.digital.com/cd/000TOOLS/ALPHA/> et Vaxen: <http://www.openvms.digital.com/cd/000TOOLS/VAX/>).

`gzip` and `tar` se trouvent aussi chez <ftp://ftp.digital.com/pub/VMS>.

Remarquez que les `gzip` et `gunzip` de GNU ne sont pas identiques à ceux de Info-ZIP. Alors que le premier n'est qu'un outil de compression et décompression, le second permet de créer des archives multi-fichiers.

## B. DÉBALLER

Si vous utilisez `VMStar`:

```
VMStar xf Your-Module.tar
```

Ou, si vous aimez bien la syntaxe de VMS :

```
tar/extract/verbose Your_Module.tar
```

#### C. CONSTRUIRE

Assurez-vous que vous êtes en possession de MMS (de Digital) ou du logiciel libre MMK (disponible chez Mad-Goat, <http://www.madgoat.com>). Puis, exécutez la commande qui suit pour créer le DESCRIP.MMS correspondant au module:

```
perl Makefile.PL
```

Vous êtes alors prêt à construire :

```
mms
```

```
mms test
```

Substituez `mmk` à `mms` ci-dessus si vous utilisez MMK.

#### D. INSTALLER

Exécutez :

```
mms install
```

Substituez `mmk` à `mms` ci-dessus si vous utilisez MMK.

#### – Si vous êtes sur MVS,

Enregistrez en binaire le fichier `.tar.gz` sur un HFS ; NE traduisez pas de ASCII à EBCDIC.

##### A. DECOMPRIMER

Decomprimez le fichier avec la commande `C<gzip -d votremodule.tar.gz>`

`gzip` se trouve notamment S<à :>

<http://www.s390.ibm.com/products/oe/bpxqp1.html>.

##### B. DÉBALLER

Déballez le résultat avec la commande :

```
pax -o to=IBM-1047,from=ISO8859-1 -r < votremodule.tar
```

Les étapes CONSTRUIRE et INSTALLER sont identiques à celles pour Unix. Certains modules génèrent des makefiles qui marchent mieux avec le make de GNU, disponible à l'adresse <http://www.mks.com/s390/gnu/index.htm>.

## 53.2 OYEZ !

Si vous avez des suggestions de modification de cette page, faites-le moi savoir. S'il vous plaît, ne m'envoyez pas de courriels de demande d'aide à l'installation de vos modules. Il y a bien trop de modules, et bien trop peu de Orwant pour que je puisse répondre à ni accuser réception de vos questions. Écrivez plutôt à l'auteur du module en question ou envoyez un message à [comp.lang.perl.modules](mailto:comp.lang.perl.modules), ou bien demandez de l'aide à quelqu'un familiarisé avec Perl sur votre système d'exploitation.

## 53.3 AUTEUR

Jon Orwant

[orwant@tpj.com](mailto:orwant@tpj.com)

The Perl Journal Perl, <http://tpj.com>

Avec l'aide incalculable de Brandon Allbery, Charles Bailey, Graham Barr, Dominic Dunlop, Jarkko Hietaniemi, Ben Holzman, Tom Horsley, Nick Ing-Simmons, Tuomas J. Lukka, Laszlo Molnar, Chris Nandor, Alan Olsen, Peter Prymmer, Gurusamy Sarathy, Christoph Spalinger, Dan Sugalski, Larry Virden, et Ilya Zakharevich.

Le 22 juillet 1998.

## 53.4 COPYRIGHT

Copyright (C) 1998 Jon Orwant. Tous droits réservés. Yves Maniette [Yves@Maniette.com](mailto:Yves@Maniette.com) pour la version en français.

Permission est accordée de fabriquer et distribuer des copies verbatim de cette documentation à condition que soient indiquées sur toutes les copies la notice de copyright et cette notice de permission.

Permission est accordée de copier et distribuer des versions modifiées de cette documentation sous les mêmes conditions que pour les copies verbatim, et à la condition qu'il soit clairement mentionné qu'il s'agit de versions modifiées, que ne soient pas modifiées les noms et titres des auteurs (bien que puissent être ajoutés des sous-titres et des noms d'auteurs supplémentaires), et que le nouveau produit soit aussi distribué selon les termes d'une notice de permission identique à celle-ci.

Permission est accordée de copier et distribuer des traductions de cette documentation en d'autres langues, dans les conditions indiquées ci-dessus pour les versions modifiées.



## **53.5 TRADUCTION**

### **53.5.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **53.5.2 Traducteur**

Yves Maniette <Yves@Maniette.com>.

### **53.5.3 Relecture**

Personne pour l'instant.

# Chapitre 54

## perlutil

Utilitaires livrés avec la distribution de Perl

### 54.1 DESCRIPTION

En plus de l'interpréteur Perl lui-même, la distribution perl installe une série d'outils sur votre système. Il y a aussi plusieurs outils qui sont utilisés par la distribution Perl elle-même dans le cadre de la procédure d'installation. Ce document a été créé afin de lister l'ensemble de ces outils, d'expliquer ce qu'ils sont et ce qu'ils font, et de fournir des pointeurs vers chaque documentation de module lorsque c'est approprié.

#### 54.1.1 DOCUMENTATION

##### **perldoc**

La principale interface de consultation de la documentation de perl est *perldoc*, quoique si vous lisez ceci c'est sûrement que vous l'avez déjà trouvée. *perldoc* extrait et met en forme la documentation d'un fichier du répertoire courant, d'un module Perl installé sur le système ou d'une page de la documentation standard, comme celle-ci. Utilisez *perldoc* <name> pour obtenir l'information à propos d'un outil décrit dans ce document.

##### **pod2man et pod2text**

S'il est lancé depuis un terminal, *perldoc* va généralement utiliser *pod2man* pour traduire du POD (Plain Old Documentation - cf. *perlpod* pour plus d'information) en une page de manuel, puis lancer *man* pour l'afficher ; si *man* n'est pas disponible, *pod2text* sera utilisé à la place et la sortie sera redirigée vers votre afficheur favori.

##### **pod2html et pod2latex**

En plus des deux convertisseurs cités précédemment, il y a deux autres convertisseurs : *pod2html*, qui produit des pages HTML depuis POD, et *pod2latex*, qui produit des fichiers LaTeX.

##### **pod2usage**

Si vous voulez juste savoir comment utiliser les outils décrits ici, *pod2usage* extrait juste la section "USAGE" ; certains outils appellent automatiquement *pod2usage* sur eux-mêmes lorsque vous les appelez avec *-help*.

##### **podselect**

*pod2usage* est une utilisation spécifique de *podselect*, un outil pour extraire des sections nommées des documents écrits en POD. Par exemple, de la même façon que les outils ont une section "USAGE", les modules Perl ont généralement une section "SYNOPSIS" : *podselect -s "SYNOPSIS" ...* : extrait cette section d'un fichier donné.

##### **podchecker**

Si vous écrivez votre propre documentation en POD, l'outil *podchecker* cherchera les erreurs dans votre marquage.

##### **splain**

*splain* est une interface d'accès à *perldiag* - donnez-lui l'un de vos messages d'erreur et il vous l'expliquera.

##### **roffitall**

L'outil *roffitall* n'est pas installé sur votre système mais réside dans le répertoire *pod/* des sources de Perl ; il convertit toute la documentation de la distribution vers le format *\*roff* pour générer finalement un ensemble de fichiers PostScript ou de fichiers texte.

### 54.1.2 CONVERTISSEURS

Pour vous aider à convertir vos programmes existants en Perl, nous avons inclus trois filtres de conversion :

#### a2p

*a2p* convertit des scripts *awk* en programmes Perl ; par exemple, *a2p -F* exécuté sur le simple script *awk {print \$2}* produira un programme Perl basé sur ce code :

```
while (<>) {
 ($F1,$F2) = split(/[:\n]/, $_, 9999);
 print $F2;
}
```

#### s2p

De la même façon, *s2p* convertit des scripts *sed* en programmes Perl. *s2p* exécuté sur *s/foo/bar* produira un programme Perl basé sur ce code :

```
while (<>) {
 chomp;
 s/foo/bar/g;
 print if $printit;
}
```

#### find2perl

Enfin, *find2perl* traduit des commandes *find* en leur équivalent Perl qui utilise le module `File::Find`. Par exemple, *find2perl . -user root -perm 4000 -print* produit le callback suivant pour `File::Find` :

```
sub wanted {
 my ($dev,$ino,$mode,$nlink,$uid,$gid);
 (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)) &&
 $uid == $uid{'root'}) &&
 (($mode & 0777) == 04000);
 print("$name\n");
}
```

Tout comme ces filtres convertissant depuis d'autres langages, l'outil *pl2pm* vous aide à convertir des bibliothèques écrites dans l'ancien style Perl 4 en modules écrits dans le style Perl5.

### 54.1.3 Administration

#### libnetcfg

Pour afficher et changer la configuration de libnet, exécutez la commande *libnetcfg*.

### 54.1.4 Développement

Il y a un ensemble d'outils qui vous aide à développer des programmes perl, et en particulier d'étendre Perl avec du C.

#### perlbug

*perlbug* est le moyen recommandé pour envoyer aux développeurs un rapport concernant un bug trouvé dans l'interpréteur perl lui-même ou dans un des modules de la bibliothèque standard ; merci de lire entièrement la documentation de *perlbug* avant de l'utiliser pour soumettre un rapport de bug.

#### h2ph

Avant l'ajout du système XS qui permet de connecter Perl avec des bibliothèques écrites en C, les programmeurs devaient récupérer les constantes d'une bibliothèque C en parcourant ses fichiers d'entêtes. Vous pourrez encore rencontrer des choses comme `require 'syscall.ph'` - le fichier *.ph* peut être créé en lançant *h2ph* sur le fichier *.h* correspondant. Lisez la documentation de *h2ph* pour savoir comment convertir un ensemble de fichiers d'entêtes en une seule fois.

#### c2ph et pstruct

*c2ph* et *pstruct*, qui sont en fait le même programme mais se comportent différemment selon la façon dont ils sont appelés, fournissent un autre moyen de travailler avec du C en Perl - ils convertissent les structures C et les déclarations d'unions en du code Perl. Cela n'est plus conseillé, *h2xs* étant maintenant préféré.

**h2xs**

*h2xs* convertit des fichiers d'entête C en modules XS en essayant de mettre en correspondance un maximum de code des modules Perl avec celui des bibliothèques C. C'est aussi très utile pour créer des squelettes de purs modules Perl.

**dprofpp**

Perl est fourni avec un profiler, le module *Devel::DProf*. L'outil *dprofpp* analyse les sorties de ce profiler et vous dit quelles sont les routines qui ont le temps d'exécution le plus long. Se référer à *Devel::DProf* pour plus d'information.

**perlcc**

*perlcc* est l'interface de la suite de compilation expérimentale de Perl.

**54.1.5 VOIR AUSSI**

*perldoc*, *pod2man*, *perlpod*, *pod2html*, *pod2usage*, *podselect*, *podchecker*, *splain*, *perldiag*, *roffitall*, *a2p*, *s2p*, *find2perl*, *File::Find*, *pl2pm*, *perlbug*, *h2ph*, *c2ph*, *h2xs*, *dprofpp*, *Devel::DProf*, *perlcc*.

**54.2 TRADUCTION****54.2.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

**54.2.2 Traducteur**

Denis Dordoigne.

**54.2.3 Relecture**

Aucune pour l'instant.

## **Quatrième partie**

# **Implémentation et interface avec le langage C**

# Chapitre 55

## perlembded

Utiliser Perl dans vos programmes en C ou C++

### 55.1 DESCRIPTION

#### 55.1.1 PRÉAMBULE

Désirez-vous :

**Utiliser du C à partir de Perl ?**

Consultez *perlxstut*, *perlxs*, *h2xs*, et *perlguts*.

**Utiliser un programme Unix à partir de Perl ?**

Consultez les paragraphes sur les BACK-QUOTES et les fonctions `system` et `exec` dans *perlfunc*.

**Utiliser du Perl à partir de Perl ?**

Consultez `do in` *perlfunc*, `eval in` *perlfunc*, `require in` *perlfunc* et `use in` *perlfunc*.

**Utiliser du C à partir du C ?**

Revoyez votre analyse.

**Utiliser du Perl à partir du C ?**

Continuez la lecture de ce document...

#### 55.1.2 SOMMAIRE

Compiler votre programme C

Ajouter un interpréteur Perl à votre programme C

Appeler un sous-programme Perl à partir de votre programme C

Évaluer un expression Perl à partir de votre programme C

Effectuer des recherches de motifs et des substitutions à partir de votre programme C

Trifouiller la pile Perl à partir de votre programme C

Maintenir un interpréteur persistant

Maintenir de multiples instances d'interpréteur

Utiliser des modules Perl utilisant les bibliothèques C, à partir de votre programme C

Intégrer du Perl sous Win32

### 55.1.3 Compiler votre programme C

Si vous avez des problèmes pour compiler les scripts de cette documentation, vous n'êtes pas le seul. La règle principale : COMPILER LES PROGRAMMES EXACTEMENT DE LA MÊME MANIÈRE QUE VOTRE PERL A ÉTÉ COMPILÉ. (désolé de hurler.)

De plus, tout programme C utilisant Perl doit être lié à la *bibliothèque perl*. Qu'est-ce ? Perl est lui-même écrit en C ; la bibliothèque perl est une collection de programmes C qui ont été utilisés pour créer votre exécutable perl (*/usr/bin/perl* ou équivalent). (Corollaire : vous ne pouvez pas utiliser Perl à partir de votre programme C à moins que Perl n'ait été compilé sur votre machine, ou installé proprement – c'est pourquoi vous ne devez pas copier l'exécutable de Perl de machine en machine sans copier aussi le répertoire *lib*.)

Quand vous utilisez Perl à partir du C, votre programme C –habituellement– allouera, « exécutera » et désallouera un objet *PerlInterpreter*, qui est défini dans la bibliothèque perl.

Si votre exemplaire de Perl est suffisamment récent pour contenir ce document (version 5.002 ou plus), alors la bibliothèque perl (et les en-têtes *EXTERN.h* et *perl.h*, dont vous aurez aussi besoin) résideront dans un répertoire qui ressemble à :

```
/usr/local/lib/perl5/votre_architecture_ici/CORE
```

ou peut-être juste

```
/usr/local/lib/perl5/CORE
```

ou encore quelque chose comme

```
/usr/opt/perl5/CORE
```

Pour avoir un indice sur l'emplacement de CORE, vous pouvez exécuter :

```
perl -MConfig -e 'print $Config{archlib}'
```

Voici comment compiler un exemple du prochain paragraphe, Ajouter un interpréteur Perl à votre programme C, sur ma machine Linux :

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

(Le tout sur une seule ligne.) Sur ma DEC Alpha utilisant une vieille version 5.003\_05, l'incantation est un peu différente :

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

Comment savoir ce qu'il faut ajouter ? Dans l'hypothèse où votre Perl est postérieur à la version 5.001, exécutez la commande `perl -v` et regardez les valeurs des paramètres "cc" et "ccflags".

Vous aurez à choisir le compilateur approprié (*cc*, *gcc*, etc...) pour votre machine : `perl -MConfig -e 'print $Config{cc}'` vous indiquera ce qu'il faut utiliser.

Vous aurez aussi à choisir le répertoire de bibliothèque approprié (*/usr/local/lib/...*) pour votre machine. Si votre ordinateur se plaint que certaines fonctions ne sont pas définies, ou qu'il ne peut trouver *-lperl*, vous devrez alors changer le chemin à l'aide de l'option `-L`. S'il se plaint qu'il ne peut trouver *EXTERN.h* et *perl.h*, vous devrez alors changer le chemin des en-têtes à l'aide de l'option `-I`.

Vous pouvez avoir besoin de bibliothèques supplémentaires. Lesquelles ? Peut-être celles indiquées par

```
perl -MConfig -e 'print $Config{libs}'
```

Si votre binaire perl est correctement installé et configuré, le module **ExtUtils::Embed** pourra déterminer toutes ces informations pour vous :

```
% cc -o interp interp.c 'perl -MExtUtils::Embed -e ccopts -e ldopts'
```

Si le module **ExtUtils::Embed** ne fait pas partie de votre distribution Perl, vous pouvez le récupérer à partir de <http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils/Embed>. Si cette documentation est livrée avec votre distribution Perl, c'est que vous utilisez la version 5.004 ou plus et vous l'avez sûrement.)

Le kit **ExtUtils::Embed** du CPAN contient aussi l'ensemble des sources des exemples de ce document, des tests, des exemples additionnels et d'autres informations qui pourraient vous servir.

### 55.1.4 Ajouter un interpréteur Perl à votre programme C

Dans un sens perl (le programme C) est un bon exemple d'intégration de Perl (le langage), donc je vais démontrer l'intégration avec *miniperlmain.c*, inclus dans les sources de la distribution. Voici une version bâtarde, non portable de *miniperlmain.c* contenant l'essentiel de l'intégration :

```
#include <EXTERN.h> /* from the Perl distribution */
#include <perl.h> /* from the Perl distribution */

static PerlInterpreter *my_perl; /*** The Perl interpreter ***/

int main(int argc, char **argv, char **env)
{
 my_perl = perl_alloc();
 perl_construct(my_perl);
 perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
 perl_run(my_perl);
 perl_destruct(my_perl);
 perl_free(my_perl);
}
```

Remarquez que nous n'utilisons pas le pointeur *env*. Normalement passé comme dernier argument de *perl\_parse*, *env* est remplacé ici par *NULL*, qui indique que l'environnement courant sera utilisé.

Compilons maintenant ce programme (je l'appellerai *interp.c*) :

```
% cc -o interp interp.c 'perl -MExtUtils::Embed -e ccopts -e ldopts'
```

Après une compilation réussie, vous pourrez utiliser *interp* comme vous le feriez de perl lui-même :

```
% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089
```

ou

```
% interp -e 'printf("%x", 3735928559)'
deadbeef
```

Vous pouvez aussi lire et exécuter des expressions Perl à partir d'un fichier en plein milieu de votre programme C, en plaçant le nom de fichier dans *argv[1]* avant d'appeler *perl\_run*.

### 55.1.5 Appeler un sous-programme Perl à partir de votre programme C

Pour appeler un sous-programme perl isolé, vous pouvez utiliser les fonctions *perl\_call\_\** décrite dans *perlcalls*. Dans cet exemple nous allons utiliser *perl\_call\_argv*.

Ceci est montré ci-dessous, dans un programme appelé *showtime.c*.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
 char *args[] = { NULL };
 my_perl = perl_alloc();
 perl_construct(my_perl);
```



```

perl_parse(my_perl, NULL, argc, argv, NULL);

/** n'utilise pas perl_run() **/

perl_call_argv("showtime", G_DISCARD | G_NOARGS, args);

perl_destruct(my_perl);
perl_free(my_perl);
}

```

où *showtime* est un sous-programme qui ne prend aucun argument (c'est le *G\_NOARGS*) et dont on ignore la valeur de retour (c'est le *G\_DISCARD*). Ces drapeaux, et les autres, sont décrits dans *perlcall*.

Je vais définir le sous-programme *showtime* dans un fichier appelé *showtime.pl* :

```

print "Je ne devrai pas etre affiche.";

sub showtime {
 print time;
}

```

Suffisamment simple. Maintenant compilez et exécutez :

```

% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% showtime showtime.pl
818284590

```

Indique le nombre de secondes écoulées depuis le 1er janvier 1970 (le début de l'ère unix), et le moment où j'ai commencé à écrire cette phrase.

Dans ce cas particulier nous n'avons pas besoin d'appeler *perl\_run*, mais en général il est considéré comme étant une bonne pratique de s'assurer de l'initialisation correcte du code de la bibliothèque, incluant l'exécution de toutes les méthodes DESTROY des objets et des blocs END {} des packages.

Si vous voulez passer des arguments aux sous-programmes Perl, vous pouvez ajouter des chaînes de caractères à la liste *args* terminée par NULL passée à *perl\_call\_argv*. Pour les autres types de données, ou pour consulter les valeurs de retour, vous devrez manipuler la pile Perl. Ceci est expliqué dans le dernier paragraphe de ce document : Trifouiller la pile Perl à partir de votre programme C.

### 55.1.6 Évaluer un expression Perl à partir de votre programme C

Perl fournit deux fonctions de l'API pour évaluer des portions de code Perl. Ce sont *perl\_eval\_sv* in *perlvars* et *perl\_eval\_pv* in *perlvars*.

Ce sont sans doute les seules fonctions que vous aurez à utiliser pour exécuter des bouts de code Perl à partir de votre programme en C. Votre code peut être aussi long que vous désirez ; il peut contenir de nombreuses expressions ; il peut utiliser *use* in *perlfunc*, *require* in *perlfunc*, et *do* in *perlfunc* d'autres fichiers Perl.

*perl\_eval\_pv* permet d'évaluer des expressions Perl, et extraire les variables pour les transformer en types C. Le programme suivant *string.c*, exécute trois chaînes Perl, extrait un *int* de la première, un <float> de la seconde et un *char \** de la troisième.

```

#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
 char *embedding[] = { "", "-e", "0" };

```

```

my_perl = perl_alloc();
perl_construct(my_perl);

perl_parse(my_perl, NULL, 3, embedding, NULL);
perl_run(my_perl);

/** Traite $a comme un entier **/
perl_eval_pv("$a = 3; $a **= 2", TRUE);
printf("a = %d\n", SvIV(perl_get_sv("a", FALSE)));

/** Traite $a comme un flottant **/
perl_eval_pv("$a = 3.14; $a **= 2", TRUE);
printf("a = %f\n", SvNV(perl_get_sv("a", FALSE)));

/** Traite $a comme une chaine **/
perl_eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
printf("a = %s\n", SvPV(perl_get_sv("a", FALSE), PL_na));

perl_destruct(my_perl);
perl_free(my_perl);
}

```

Toutes les fonctions étranges comportant *sv* dans leurs noms aident à convertir les scalaires Perl en types C. Elles sont décrites dans *perlguts*.

Si vous compilez et exécutez *string.c*, vous pourrez voir les résultats de l'utilisation de *SvIV()* pour créer un *int*, *SvNV()* pour créer un *float* et *SvPV()* pour créer une chaîne :

```

a = 9
a = 9.859600
a = Just Another Perl Hacker

```

Dans l'exemple ci-dessus, nous avons créé une variable globale pour stocker temporairement la valeur calculée par nos expressions eval. Il est aussi possible et c'est dans la plupart des cas la meilleure stratégie de récupérer la valeur de retour à partir de *perl\_eval\_pv()* à la place. Exemple :

```

...
SV *val = perl_eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val, PL_na));
...

```

De cette manière, nous évitons la pollution de l'espace des noms en ne créant pas de variables globales et nous avons aussi simplifié notre code.

### 55.1.7 Effectuer des recherches de motifs et des substitutions à partir de votre programme C

La fonction *perl\_eval\_sv()* nous permet d'évaluer des bouts de code Perl, nous pouvons donc définir quelques fonctions qui l'utilisent pour créer des fonctions « spécialisées » dans les recherches et substitutions : *match()*, *substitute()*, et *matches()*.

```
I32 match(SV *string, char *pattern);
```

Étant donné une chaîne et un motif (ex., *m/clasp/* ou */\b\w\*\b/*, qui peuvent apparaître dans votre programme C comme *"/\b\w\*\b/"*), *match()* retourne 1 Si la chaîne correspond au motif et 0 autrement.

```
int substitute(SV **string, char *pattern);
```

Étant donné un pointeur vers un SV et une opération =~ (ex., *s/bob/robert/g* ou *tr[A-Z][a-z]*), *substitute()* modifie la chaîne à l'intérieur de l'AV en suivant l'opération, retournant le nombre de substitutions effectuées.

```
int matches(SV *string, char *pattern, AV **matches);
```

Étant donné un SV, un motif et un pointeur vers un AV vide, `matches()` évalue `$string =~ $pattern` dans un contexte de tableau et remplit `matches` avec les éléments du tableau, retournant le nombre de correspondances trouvées.

Voici un exemple, `match.c`, qui les utilise tous les trois :

```
#include <EXTERN.h>
#include <perl.h>

/** my_perl_eval_sv(code, error_check)
** une sorte de perl_eval_sv(),
** mais nous retirons la valeur de retour de la pile.
**/
SV* my_perl_eval_sv(SV *sv, I32 croak_on_error)
{
 dSP;
 SV* retval;

 PUSHMARK(SP);
 perl_eval_sv(sv, G_SCALAR);

 SPAGAIN;
 retval = POPs;
 PUTBACK;

 if (croak_on_error && SvTRUE(ERRSV))
 croak(SvPVx(ERRSV, PL_na));

 return retval;
}

/** match(chaine, motif)
**
** Utilise pour faire des recherches dans un contexte scalaire.
**
** Retourne 1 si la recherche est reussie; 0 autrement.
**/
I32 match(SV *string, char *pattern)
{
 SV *command = NEWSV(1099, 0), *retval;

 sv_setpvf(command, "my $string = '%s'; $string =~ %s",
 SvPV(string, PL_na), pattern);

 retval = my_perl_eval_sv(command, TRUE);
 SvREFCNT_dec(command);

 return SvIV(retval);
}

/** substitute(chaine, motif)
**
** Utilisee pour les operations =~ qui modifient leur membre de gauche (s/// and tr///)
**
** Retourne le nombre de remplacement effectue
** Note: cette fonction modifie la chaine.
**/
```

```

I32 substitute(SV **string, char *pattern)
{
 SV *command = NEWSV(1099, 0), *retval;

 sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
 SvPV(*string, PL_na), pattern);

 retval = my_perl_eval_sv(command, TRUE);
 SvREFCNT_dec(command);

 *string = perl_get_sv("string", FALSE);
 return SvIV(retval);
}

/** matches(chaine, motifs, correspondances)
**
** Utilise pour faire des recherches dans un contexte tableau.
**
** Retourne le nombre de correspondance
** et remplis le tableau correspondance avec les sous-chaines trouvees.
**/

I32 matches(SV *string, char *pattern, AV **match_list)
{
 SV *command = NEWSV(1099, 0);
 I32 num_matches;

 sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
 SvPV(string, PL_na), pattern);

 my_perl_eval_sv(command, TRUE);
 SvREFCNT_dec(command);

 *match_list = perl_get_av("array", FALSE);
 num_matches = av_len(*match_list) + 1; /** assume $[is 0 **/

 return num_matches;
}

main (int argc, char **argv, char **env)
{
 PerlInterpreter *my_perl = perl_alloc();
 char *embedding[] = { "", "-e", "0" };
 AV *match_list;
 I32 num_matches, i;
 SV *text = NEWSV(1099, 0);

 perl_construct(my_perl);
 perl_parse(my_perl, NULL, 3, embedding, NULL);

 sv_setpv(text, "When he is at a convenience store and the bill comes to some amount like 76 cents,");

 if (match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
 printf("match: Text contains the word 'quarter'.\n\n");
 else
 printf("match: Text doesn't contain the word 'quarter'.\n\n");

 if (match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
 printf("match: Text contains the word 'eighth'.\n\n");
 else
 printf("match: Text doesn't contain the word 'eighth'.\n\n");
}

```

```

/** Trouve toutes les occurrences de /wi../ **/
num_matches = matches(text, "m/(wi..)/g", &match_list);
printf("matches: m/(wi..)/g found %d matches...\n", num_matches);

for (i = 0; i < num_matches; i++)
 printf("match: %s\n", SvPV(*av_fetch(match_list, i, FALSE), PL_na));
printf("\n");

/** Retire toutes les voyelles de text **/
num_matches = substitute(&text, "s/[aeiou]//gi");
if (num_matches) {
 printf("substitute: s/[aeiou]//gi...%d substitutions made.\n",
 num_matches);
 printf("Now text is: %s\n\n", SvPV(text, PL_na));
}

/** Tente une substitution **/
if (!substitute(&text, "s/Perl/C/")) {
 printf("substitute: s/Perl/C...No substitution made.\n\n");
}

SvREFCNT_dec(text);
PL_perl_destruct_level = 1;
perl_destruct(my_perl);
perl_free(my_perl);
}

```

Affiche (les lignes trop longues ont été coupées) :

```

match: Text contains the word 'quarter'.

match: Text doesn't contain the word 'eighth'.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]//gi...139 substitutions made.
Now text is: Whn h s t cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt bck
qrtr, bt h hs n d *wht*. H fmbles thrgh hs rd sqzy chngprs nd gvs th by
thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crct mnt. Th by gvs
hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s hs prz. -RCHH

substitute: s/Perl/C...No substitution made.

```

### 55.1.8 Trifouiller la pile Perl à partir de votre programme C

Dans la plupart des livres d'informatique, les piles sont expliquées à l'aide de quelque chose comme une pile d'assiettes de cafétéria : la dernière chose que vous avez posée sur la pile est la première que vous allez en retirer. Ça correspond à nos buts : votre programme C déposera des arguments sur la "pile Perl", fermera ses yeux pendant que quelque chose de magique se passe, et retirera le résultat –la valeur de retour de votre sous-programme Perl– de la pile.

Premièrement, vous devez savoir comment convertir les types C en types Perl et inversement, en utilisant `newSViv()`, `sv_setnv()`, `newAV()` et tous leurs amis. Elles sont décrites dans *perlguts*.

Ensuite vous avez besoin de savoir comment manipuler la pile Perl. C'est décrit dans *perlcall*.

Une fois que vous avez compris ceci, intégré du Perl en C est facile.

Parce que le C ne dispose pas de fonction prédéfinie pour calculer une puissance entière, rendons l'opérateur Perl `**` disponible (ceci est moins utile que ça en a l'air, car Perl implémente l'opérateur `**` à l'aide de la fonction C `pow()`). Premièrement je vais créer une souche de fonction d'exponentiation dans *power.pl* :

```

sub expo {
 my ($a, $b) = @_;
 return $a ** $b;
}

```

Maintenant je vais écrire un programme C, *power.c*, avec une fonction *PerlPower()* qui contient tous les perlputs nécessaires pour déposer les deux arguments dans *expo()* et récupérer la valeur de retour. Prenez une grande respiration...

```

#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

static void
PerlPower(int a, int b)
{
 dSP; /* initialise le pointeur de pile */
 ENTER; /* tout ce qui est cree a partir d'ici */
 SAVETMPS; /* ... est une variable temporaire */
 PUSHMARK(SP); /* sauvegarde du pointeur de pile */
 XPUSHs(sv_2mortal(newSViv(a))); /* depose la base dans la pile */
 XPUSHs(sv_2mortal(newSViv(b))); /* depose l'exposant dans la pile */
 PUTBACK; /* rend global le pointeur local de pile */
 perl_call_pv("expo", G_SCALAR); /* appelle la fonction */
 SPAGAIN; /* rafraichit le pointeur de pile */
 /* retire la valeur de retour de la pile */
 printf ("%d to the %dth power is %d.\n", a, b, POPi);
 PUTBACK;
 FREETMPS; /* libere la valeur de retour */
 LEAVE; /* ...et retire les arguments empiles */
}

int main (int argc, char **argv, char **env)
{
 char *my_argv[] = { "", "power.pl" };

 my_perl = perl_alloc();
 perl_construct(my_perl);

 perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
 perl_run(my_perl);

 PerlPower(3, 4); /*** Calcule 3 ** 4 ***/

 perl_destruct(my_perl);
 perl_free(my_perl);
}

```

Compiler et exécuter :

```

% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% power
3 to the 4th power is 81.

```

### 55.1.9 Maintenir un interpréteur persistant

Lorsque l'on développe une application interactive et/ou potentiellement de longue durée, c'est une bonne idée de maintenir un interpréteur persistant plutôt que d'allouer et de construire un nouvel interpréteur de nombreuses fois. La raison principale est la vitesse : car Perl ne sera alors chargé qu'une seule fois en mémoire.

De toutes façons, vous devez être plus prudent avec l'espace des noms et la portée des variables lorsque vous utilisez un interpréteur persistant. Dans les exemples précédents nous utilisions des variables globales dans le package par défaut `main`. Nous savions exactement quel code sera exécuté, et supposons que nous pourrions éviter les collisions de variables et une extension atroce de la table des symboles.

Supposons que notre application est un serveur qui exécutera occasionnellement le code Perl de quelques fichiers arbitraires. Notre serveur n'a plus moyen de savoir quel code il va exécuter. C'est très dangereux.

Si le fichier est fourni à `perl_parse()`, compilé dans un interpréteur nouvellement créé, et subséquentement détruit par `perl_destruct()` après, vous êtes protégés de la plupart des problèmes d'espace de nom.

Une manière d'éviter les collisions d'espace de nom dans ce cas est de transformer le nom de fichier en un nom de package garanti unique, et de compiler le code de ce package en utilisant `eval` in *perlfunc*. Dans l'exemple ci-dessous, chaque fichier ne sera compilé qu'une seule fois. Ou l'application peut choisir de nettoyer la table des symboles associée au fichier dès qu'il n'est plus nécessaire. En utilisant `perl_call_argv` in *perlcall*, nous allons appeler le sous-programme `Embed::Persistent::eval_file` contenu dans le fichier `persistent.pl` et lui passer le nom de fichier et le booléen `nettoyer/cacher` comme arguments.

Notez que le processus continuera de grossir pour chaque fichier qu'il utilisera. De plus, il peut y avoir des sous-programmes `AUTOLOAD` et d'autres conditions qui peuvent faire que la table de symboles Perl grossisse. Vous pouvez vouloir ajouter un peu de logique qui surveille la taille du processus, ou qui redémarre tout seul après un certain nombre de requêtes pour être sûr que la consommation mémoire est minimisée. Vous pouvez aussi vouloir limiter la portée de vos variables autant que possible grâce à `my` in *perlfunc*.

```
package Embed::Persistent;
#persistent.pl

use strict;
use vars '%Cache';
use Symbol qw(delete_package);

sub valid_package_name {
 my($string) = @_;
 $string =~ s/([^\A-Za-z0-9\])/sprintf("%2x",unpack("C",$1))/eg;
 # Seconde passe pour les mots commençant par un chiffre.
 $string =~ s|/(\d)|sprintf("/%2x",unpack("C",$1))|eg;

 # Le transformer en nom de package reel
 $string =~ s|/|::|g;
 return "Embed" . $string;
}

sub eval_file {
 my($filename, $delete) = @_;
 my $package = valid_package_name($filename);
 my $mtime = -M $filename;
 if(defined $Cache{$package}{mtime}
 &&
 $Cache{$package}{mtime} <= $mtime)
 {
 # nous avons déjà compilé ce sous-programme,
 # il n'a pas été mis-à-jour sur le disque, rien à faire
 print STDERR "already compiled $package->handler\n";
 }
 else {
 local *FH;
 open FH, $filename or die "open '$filename' $!";
 local($/) = undef;
 my $sub = <FH>;
 close FH;
 }
}
```

```

#encadre le code dans un sous-programme de notre package unique
my $eval = qq{package $package; sub handler { $sub; }};
{
 # cacher nos variables dans ce bloc
 my($filename,$mtime,$package,$sub);
 eval $eval;
}
die $@ if $@;

le mettre en cache a moins qu'on le detruit a chaque fois
$Cache{$package}{mtime} = $mtime unless $delete;
}

eval {$package->handler;};
die $@ if $@;

delete_package($package) if $delete;

#Si vous voulez voir ce qui se passe
#print Devel::Symdump->rnew($package)->as_string, $/;
}

1;

__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = Detruire la table des symboles du fichier apres chaque requete, 0 = ne pas le faire */
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

static PerlInterpreter *perl = NULL;

int
main(int argc, char **argv, char **env)
{
 char *embedding[] = { "", "persistent.pl" };
 char *args[] = { "", DO_CLEAN, NULL };
 char filename [1024];
 int exitstatus = 0;

 if((perl = perl_alloc()) == NULL) {
 fprintf(stderr, "no memory!");
 exit(1);
 }
 perl_construct(perl);

 exitstatus = perl_parse(perl, NULL, 2, embedding, NULL);

 if(!exitstatus) {
 exitstatus = perl_run(perl);

 while(printf("Enter file name: ") && gets(filename)) {

```



```

/* appeler le sous-programme, passer son nom de fichier en argument */
args[0] = filename;
perl_call_argv("Embed::Persistent::eval_file",
 G_DISCARD | G_EVAL, args);

/* Verifier $@ */
if(SvTRUE(ERRSV))
 fprintf(stderr, "eval error: %s\n", SvPV(ERRSV,PL_na));
}
}

PL_perl_destruct_level = 0;
perl_destruct(perl);
perl_free(perl);
exit(exitstatus);
}

```

Compilons :

```
% cc -o persistent persistent.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Voici un exemple de fichier script :

```

#test.pl
my $string = "hello";
foo($string);

sub foo {
 print "foo says: @_ \n";
}

```

Exécutons :

```

% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C

```

### 55.1.10 Maintenir de multiples instances d'interpréteur

Quelques rares applications nécessitent de créer plus d'un interpréteur lors d'une session. Une telle application peut décider sporadiquement de libérer toutes les ressources associées à l'interpréteur.

Le programme doit s'assurer que ça ait lieu *avant* qu'un nouvel interpréteur soit construit. Par défaut, la variable globale `PL_perl_destruct_level` est positionnée à 0, puisqu'un nettoyage supplémentaire n'est pas nécessaire lorsqu'un programme n'utilise qu'un seul interpréteur.

Positionner `PL_perl_destruct_level` à 1 rend tout plus propre :

```

PL_perl_destruct_level = 1;

while(1) {
 ...
 /* repositionner les variables globales avec PL_perl_destruct_level = 1 */
 perl_construct(my_perl);
 ...
 /* Nettoie et remet a zero _tout_ pendant perl_destruct */
 perl_destruct(my_perl);
 perl_free(my_perl);
 ...
 /* Reconnencons encore et encore ! */
}

```

Lorsque `perl_destruct()` est appelé, l'arbre d'analyse syntaxique et les tables de symboles de l'interpréteur sont nettoyées, et les variables globales sont repositionnées.

Maintenant supposons que nous ayons plus d'une instance d'interpréteur s'exécutant en même temps. Ceci est faisable, mais seulement si le drapeau `-MULTIPLICITY` a été utilisé lors de la compilation de Perl. Par défaut, cela positionne `PL_perl_destruct_level` à 1.

Essayons :

```
#include <EXTERN.h>
#include <perl.h>

/* nous allons integrer deux interpreteurs */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
 PerlInterpreter
 *one_perl = perl_alloc(),
 *two_perl = perl_alloc();
 char *one_args[] = { "one_perl", SAY_HELLO };
 char *two_args[] = { "two_perl", SAY_HELLO };

 perl_construct(one_perl);
 perl_construct(two_perl);

 perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
 perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

 perl_run(one_perl);
 perl_run(two_perl);

 perl_destruct(one_perl);
 perl_destruct(two_perl);

 perl_free(one_perl);
 perl_free(two_perl);
}
```

Compilez comme d'habitude :

```
% cc -o multiplicity multiplicity.c 'perl -MExtUtils::Embed -e ccopts -e ldopts'
```

Exécutons, exécutons :

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

### 55.1.11 Utiliser des modules Perl utilisant les bibliothèques C, à partir de votre programme C

Si vous avez joué avec les exemples ci-dessus et avez essayé d'intégrer un script qui utilise (`use()`) un module Perl (tel que `Socket`) qui lui-même utilise une bibliothèque C ou C++, vous avez probablement vu le message suivant :

```
Can't load module Socket, dynamic loading not available in this perl.
(You may need to build a new perl executable which either supports
dynamic loading or has the Socket module statically linked into it.)
```

{Traduction : Ne peut charger le module Socket, le chargement dynamique n'est pas disponible dans ce perl. (Vous avez peut-être besoin de compiler un nouvel exécutable perl qui supporte le chargement dynamique ou qui soit lié statiquement au module Socket.)

Quel est le problème ?

Votre interpréteur ne sait pas communiquer avec ces extensions de son propre chef. Un peu de colle l'aidera. Jusqu'à maintenant vous appelez *perl\_parse()*, en lui passant NULL comme second argument :

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

C'est là que le code de collage peut être inséré pour créer le contact initial entre Perl et les routines C/C++ liées. Jettons un coup d'oeil à *perlmain.c* pour voir comment Perl fait :

```
#ifdef __cplusplus
define EXTERN_C extern "C"
#else
define EXTERN_C extern
#endif

static void xs_init _((void));

EXTERN_C void boot_DynaLoader _((CV* cv));
EXTERN_C void boot_Socket _((CV* cv));

EXTERN_C void
xs_init()
{
 char *file = __FILE__;
 /* DynaLoader est un cas special */
 newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
 newXS("Socket::bootstrap", boot_Socket, file);
}
```

Explication : pour chaque extension liée à l'exécutable Perl (déterminé lors de sa configuration initiale sur votre ordinateur ou lors de l'ajout de nouvelles extensions), un sous-programme Perl est créé pour incorporer les routines de l'extension. Normalement, ce sous-programme est nommé *Module::bootstrap()* et est invoqué lors du *use Module*. Tour à tour, il passe par un XSUB, *boot\_Module*, qui crée un pendant pour chaque XSUB de l'extension. Ne vous inquiétez pas de cette partie ; laissez ceci aux auteurs de *xsubpp* et des extensions. Si votre extension est chargée dynamiquement, DynaLoader créé au vol *Module::bootstrap()* pour vous. En fait, si vous disposez d'un DynaLoader fonctionnant correctement, il est rarement nécessaire de lier statiquement d'autres extensions.

Une fois que vous disposez de ce code, placez-le en deuxième argument de *perl\_parse()* :

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

Puis compilez :

```
% cc -o interp interp.c 'perl -MExtUtils::Embed -e ccopts -e ldopts'

% interp
use Socket;
use SomeDynamicallyLoadedModule;

print "Maintenant je peux utiliser des extensions!\n"
```

**ExtUtils::Embed** peut aussi automatiser l'écriture du code de collage *xs\_init*.

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxsi.c
% cc -c perlxsi.c 'perl -MExtUtils::Embed -e ccopts'
% cc -c interp.c 'perl -MExtUtils::Embed -e ccopts'
% cc -o interp perlxsi.o interp.o 'perl -MExtUtils::Embed -e ldopts'
```

Consultez *perlx*s et *perlguts* pour plus de détails.

## 55.2 Intégrer du Perl sous Win32

Au moment où j'écris ceci (5.004), il existe deux versions de perl fonctionnant sous Win32. (Ces deux versions fusionnent en 5.005.) S'interfacer avec la bibliothèque Perl d'ActiveState ne se fait pas tout à fait de la même manière que dans les exemples de cette documentation, car de nombreux changements ont été effectués dans l'interface interne de programmation de Perl. Mais il est possible d'intégrer le noyau Perl d'ActiveState. Pour les détails, jetez un oeil à la FAQ Perl pour Win32 à [http://www.perl.com/perl/faq/win32/Perl\\_for\\_Win32\\_FAQ.html](http://www.perl.com/perl/faq/win32/Perl_for_Win32_FAQ.html).

Avec le Perl "officiel" version 5.004 ou plus, tous les exemples de ce document pourront être compilés et exécutés sans modification, même si le processus de compilation est plutôt différent entre Unix et Win32.

Pour commencer, les BACKTICKS ne fonctionnent pas sous l'interpréteur de commandes natif de Win32. Le kit ExtUtils::Embed du CPAN est fourni avec un script appelé **genmake**, qui génère un simple makefile pour compiler un programme à partir d'un code source C. Il peut être utilisé de cette manière :

```
C:\ExtUtils-Embed\eg> perl genmake interp.c
C:\ExtUtils-Embed\eg> nmake
C:\ExtUtils-Embed\eg> interp -e "print qq{I'm embedded in Win32!\n}"
```

Vous pouvez vouloir utiliser un environnement plus robuste tel que Microsoft Developer Studio. Dans ce cas, exécutez la commande suivante pour générer perlxi.c :

```
perl -MExtUtils::Embed -e xsinit
```

Créez un nouveau projet et Insérer -> Fichier dans le projet: perlxi.c, perl.lib, ainsi que votre propre code source, par ex. interp.c. Typiquement vous pourrez trouver perl.lib dans **C:\perl\lib\CORE**, sinon, vous pourrez trouver le répertoire **CORE** relatif à perl -V:archlib. Le studio devra aussi connaître ce chemin pour qu'il puisse trouver les fichiers d'entêtes Perl. Ce chemin peut être ajouté par le menu Tools -> Options -> Directories. Puis sélectionnez Build -> Build interp.exe et vous pourrez y aller.

## 55.3 MORALITE

Vous pouvez quelquefois *écrire du code plus rapide* en C, mais vous pourrez toujours *écrire plus rapidement du code* en Perl. Puisque vous pouvez utiliser l'un avec l'autre, combinez-les comme vous le désirez.

## 55.4 AUTEUR

Jon Orwant <[orwant@tpj.com](mailto:orwant@tpj.com)> and Doug MacEachern <[doug@osf.org](mailto:doug@osf.org)>, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobeld, and Ilya Zakharevich.

Doug MacEachern a écrit un article sur le sujet dans le Volume 1, Issue 4 du The Perl Journal (<http://tpj.com>). Doug est aussi l'auteur de l'intégration Perl la plus utilisée : mod\_perl ([perl.apache.org](http://perl.apache.org)), qui intègre Perl au serveur web Apache. Oracle, Binary Evolution, ActiveState, et le nsapi\_perl de Ben Sugars ont utilisé ce modèle pour Oracle, Netscape et les extension Perl de Internet Information Server.

July 22, 1998

## 55.5 COPYRIGHT ORIGINAL

Copyright (C) 1995, 1996, 1997, 1998 Doug MacEachern and Jon Orwant. All Rights Reserved.

Permission is granted to make and distribute verbatim copies of this documentation provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that they are marked clearly as modified versions, that the authors' names and title are unchanged (though subtitles and additional authors' names may be added), and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

## **55.6 TRADUCTION**

### **55.6.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **55.6.2 Traducteurs**

Marc Carmier <[carmier@immortels.frmug.org](mailto:carmier@immortels.frmug.org)>

### **55.6.3 Relecture**

Julien Gilles, Gérard Delafond

# Chapitre 56

## perldebguts

Les entrailles du débogage de Perl

### 56.1 DESCRIPTION

Ceci n'est pas la page de manuel `perldebug(1)`, qui vous indique comment utiliser le débogueur. Cette page donne des détails de bas niveau dont la compréhension va de difficile à impossible pour quiconque n'étant pas incroyablement intime avec les entrailles de Perl. Caveat lector (Ben me v'la frais, NDT).

### 56.2 Éléments Internes du Débogueur

Perl a des hooks spéciaux de débogage à la compilation et à l'exécution qui sont utilisés pour créer des environnements de débogage. Ces hooks ne doivent pas être confondus avec la commande `perl -Dxxx` décrite dans `perlrun`, qui n'est utilisable que si l'on utilise une version spéciale de Perl compilée selon les instructions du fichier pod `INSTALL` dans l'arborescence source de Perl (la phrase de la VO est incomplète... NDT).

Par exemple, lorsque vous appelez la fonction intégrée `caller` de Perl depuis le paquetage `DB`, les arguments avec lesquels a été appelée la frame correspondante de la pile sont copiés dans le tableau `@DB::args`. Le mécanisme général est validé en appelant Perl avec l'option `-d`, et les caractéristiques supplémentaires suivantes sont disponibles (cf. `$P` in `perlvar`) :

- Perl insère le contenu de `$ENV{PERL5DB}` (ou de `BEGIN {require 'perl5db.pl'}` en son absence) avant la première ligne de l'application.
- Le tableau `@{"_<$filename"}` est le contenu ligne à ligne de `$filename` pour tous les fichiers compilés. Même chose pour les chaînes évaluées contenant des sous-programmes, ou qui sont actuellement exécutées. Le `$filename` pour les chaînes évaluées ressemble à `(eval 34)`. Les assertions de code dans les expressions rationnelles ressemblent à `(re_eval 19)`.
- Le hachage `%{"_<$filename"}` contient les points d'arrêt et les actions (ses clés sont les numéros de lignes), et des entrées individuelles sont modifiables (par opposition au hachage tout entier). Seul `true/false` est important pour Perl, même si les valeurs utilisées par `perl5db.pl` ont la forme `"$break_condition\0$action"`. Les valeurs sont magiques dans un contexte numérique : ce sont des zéros si la ligne ne peut pas être le lieu d'un point d'arrêt. Idem pour les chaînes évaluées qui contiennent des sous-programmes, ou qui sont en cours d'exécution. Le `$filename` pour les chaînes évaluées ressemble à `(eval 34)` ou à `(re_eval 19)`.
- Le scalaire `${"_<$filename"}` contient `"_<$filename"`. Idem pour les chaînes évaluées qui contiennent des sous-programmes, ou qui sont en cours d'exécution. Le `$filename` pour les chaînes évaluées ressemble à `(eval 34)` ou à `(re_eval 19)`.
- Après la compilation de chaque fichier exigé par `require`, mais avant son exécution, `DB::postponed(*{"_<$filename"})` est appelé (si le sous-programme `DB::postponed` existe). Ici le `$filename` est le nom développé du fichier exigé, tel que trouvé dans les valeurs de `%INC`.
- Après la compilation de chaque sous-programme `subname`, l'existence de `DB::postponed{subname}` est vérifiée. Si cette clé existe, `DB::postponed(subname)` est appelé si le sous-programme `DB::postponed` existe.
- Un hachage `%DB::sub` est maintenu, dont les clés sont les noms des sous-programmes et dont les valeurs ont la forme `filename:startline-endline`. `filename` a la forme `(eval 34)` pour les sous-programmes définis dans des `evals`, ou `(re_eval 19)` pour ceux qui se trouvent dans des assertions de code d'expression rationnelle.

- Lorsque l'exécution de votre programme atteint un endroit pouvant avoir un point d'arrêt, le sous-programme `DB::DB()` est appelé si l'une des variables `$DB::trace`, `$DB::single` ou `$DB::signal` est vraie. Ces variables ne sont pas localisables. Cette caractéristique est invalidée lorsque le contrôle est à l'intérieur de `DB::DB()` ou de fonctions appelées à partir de lui (À moins que `$^D & (1<30)`) soit vrai.
- Lorsque l'exécution de l'application atteint un appel de sous-programme, un appel à `&DB::sub(args)` est réalisé à la place, avec `$DB::sub` contenant le nom du sous-programme appelé. Ceci ne se produit pas si le sous-programme ait été compilé dans le paquetage DB.

Notez que si `&DB::sub` a besoin de données externes pour son bon fonctionnement, aucun appel de sous-programme n'est possible tant que ce n'est pas fait. Pour le débogueur standard, la variable `$DB::deep` (profondeur des niveaux de récursion dans le débogueur que vous pouvez atteindre avant un arrêt obligatoire) donne un exemple de telle dépendance.

### 56.2.1 Écrire Votre Propre Débogueur

Le débogueur fonctionnel minimal consiste en une seule ligne

```
sub DB::DB {}
```

ce qui est bien pratique comme contenu de la variable d'environnement `PERL5DB` :

```
$ "PERL5DB=sub DB::DB {}" perl -d your-script
```

Un autre débogueur minimal, un petit peu plus utile, pourrait être créé, la ligne unique étant

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

Ce débogueur afficherait le nombre séquentiel d'instructions reconstruées, et attendrait que vous appuyiez sur entrée pour continuer.

Le débogueur suivant est plutôt fonctionnel :

```
{
 package DB;
 sub DB {}
 sub sub {print ++$i, " $sub\n"; &$sub}
}
```

Il affiche le nombre séquentiel d'appels de sous-programmes et le nom des sous-programmes appelés. Notez que `&DB::sub` doit être compilé dans le paquetage DB.

Au démarrage, le débogueur lit votre fichier `rc` (`./perldb` ou `~/perldb` sous Unix), qui peut définir des options importantes. Ce fichier peut définir un sous-programme `&afterinit` devant être exécuté après l'initialisation du débogueur.

Après la lecture du fichier `rc`, le débogueur lit la variable d'environnement `PERLDB_OPTS` et l'analyse comme si c'était le reste de la ligne `0 ...` dans le prompt du débogueur.

Il maintient aussi des variables internes magiques, telles que `@DB::dbline` et `%DB::dbline` qui sont des alias de `@{":_<fichier_courant"}` et `%{":_<fichier_courant"}`. Ici, `fichier_courant` est le fichier actuellement sélectionné, soit choisi explicitement par la commande `f` du débogueur, ou implicitement par le flux de l'exécution).

Certaines fonctions sont fournies pour simplifier la personnalisation. Voir *Options Configurables* in *perldebug* pour une description des options analysées par `DB::parse_options(string)`. La fonction `DB::dump_trace(skip[, count])` saute le nombre spécifié de frames et retourne une liste contenant des informations sur les frames de l'appelant (la totalité d'entre elles si `count` est manquant). Chaque entrée est une référence à un hachage contenant le contexte des clés (soit `.`, `$` ou `@`), `sub` (nom du sous-programme, ou infos sur `eval`), `args` (`undef` ou une référence à un tableau), `fichier`, et `ligne`.

La fonction `DB::print_trace(FH, skip[, count[, short]])` affiche des infos formatées sur les frames de l'appelant. Les deux dernières fonctions peuvent être pratiques comme arguments des commandes `<` et `>`.

Notez que toute variable et toute fonction qui n'est pas documentée ici (ou dans *perldebug*) est considérée comme réservée à un usage interne uniquement, et est en tant que telle sujette à changement sans préavis.

## 56.3 Exemples de Listages des Frames

L'option `frame` peut être utilisée pour contrôler la sortie des informations sur les frames. Par exemple, comparez cette trace d'une expression :

```
$ perl -de 42
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or 'h h' for help.

main::(-e:1): 0
DB<1> sub foo { 14 }

DB<2> sub bar { 3 }

DB<3> t print foo() * bar()
main::((eval 172):3): print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

avec celle-ci, une fois que l'option `frame=2` a été validée :

```
DB<4> 0 f=2
frame = '2'
DB<5> t print foo() * bar()
3: foo() * bar()
entering main::foo
2: sub foo { 14 };
exited main::foo
entering main::bar
2: sub bar { 3 };
exited main::bar
42
```

Pour les besoins de la démonstration, nous présentons ci-dessous un listage laborieux obtenu en plaçant votre variable d'environnement `PERLDB_OPTS` à la valeur `f=n N`, et en exécutant `perl -d -V` sur la ligne de commande. Les exemples utilisent diverses valeurs de `n` pour vous montrer la différence entre ces réglages. Aussi longs qu'ils puissent paraître, ils ne sont pas des listages complets, mais seulement des extraits.

### valeur 1

```
entering main::BEGIN
entering Config::BEGIN
Package lib/Exporter.pm.
Package lib/Carp.pm.
Package lib/Config.pm.
entering Config::TIEHASH
entering Exporter::import
entering Exporter::export
entering Config::myconfig
entering Config::FETCH
entering Config::FETCH
entering Config::FETCH
entering Config::FETCH
```

### valeur 2



```

entering main::BEGIN
 entering Config::BEGIN
 Package lib/Exporter.pm.
 Package lib/Carp.pm.
 exited Config::BEGIN
 Package lib/Config.pm.
 entering Config::TIEHASH
 exited Config::TIEHASH
 entering Exporter::import
 entering Exporter::export
 exited Exporter::export
 exited Exporter::import
exited main::BEGIN
entering Config::myconfig
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH

```

**valeur 4**

```

in $=main::BEGIN() from /dev/null:0
in $=Config::BEGIN() from lib/Config.pm:2
 Package lib/Exporter.pm.
 Package lib/Carp.pm.
 Package lib/Config.pm.
in $=Config::TIEHASH('Config') from lib/Config.pm:644
in $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
 in $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from li
in @=Config::myconfig() from /dev/null:0
in $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
in $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
in $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
in $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
in $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
in $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574

```

**valeur 6**

```

in $=main::BEGIN() from /dev/null:0
in $=Config::BEGIN() from lib/Config.pm:2
 Package lib/Exporter.pm.
 Package lib/Carp.pm.
out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
in $=Config::TIEHASH('Config') from lib/Config.pm:644
out $=Config::TIEHASH('Config') from lib/Config.pm:644
in $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
 in $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
 out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in @=Config::myconfig() from /dev/null:0
 in $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
 in $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574

```

**valeur 14**

```

in $=main::BEGIN() from /dev/null:0
in $=Config::BEGIN() from lib/Config.pm:2

```

```

Package lib/Exporter.pm.
Package lib/Carp.pm.
out $=Config::BEGIN() from lib/Config.pm:0
Package lib/Config.pm.
in $=Config::TIEHASH('Config') from lib/Config.pm:644
out $=Config::TIEHASH('Config') from lib/Config.pm:644
in $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
in $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/E
out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in @=Config::myconfig() from /dev/null:0
in $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
in $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574

```

**valeur 30**

```

in $=CODE(0x15eca4)() from /dev/null:0
in $=CODE(0x182528)() from lib/Config.pm:2
Package lib/Exporter.pm.
out $=CODE(0x182528)() from lib/Config.pm:0
scalar context return from CODE(0x182528): undef
Package lib/Config.pm.
in $=Config::TIEHASH('Config') from lib/Config.pm:628
out $=Config::TIEHASH('Config') from lib/Config.pm:628
scalar context return from Config::TIEHASH: empty hash
in $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
in $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporter.pm:171
out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/Exporter.pm:171
scalar context return from Exporter::export: ''
out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
scalar context return from Exporter::import: ''

```

Dans tous les cas montrés ci-dessus, l'indentation des lignes montre l'arbre d'appels. Si le bit 2 de `frame` est mis, alors une ligne est affichée aussi à la sortie d'un sous-programme. Si le bit 4 est mis, alors les arguments sont aussi affichés ainsi que les infos sur l'appelant. Si le bit 8 est mis, les arguments sont affichés même s'ils sont liés ou sont des références. Si le bit 16 est mis, la valeur de retour est aussi affichée.

Lorsqu'un paquetage est compilé, une ligne comme celle-ci

```
Package lib/Carp.pm.
```

est affichée avec l'indentation adéquate.

## 56.4 Débogage des expressions rationnelles

Il y a deux façons d'obtenir une trace de débogage pour les expressions rationnelles.

Si votre perl est compilé avec `-DDEBUGGING` activé, vous pouvez utiliser l'option de ligne de commande **-Dr**.

Sinon, vous pouvez indiquer `use re 'debug'`, qui est effectif à la fois à la compilation et lors de l'exécution. Il n'a *pas* de portée lexicale.

### 56.4.1 Trace lors de la compilation

La trace de débogage lors de la compilation a cette allure :

```

compiling RE '[bc]d(ef*g)+h[ij]k$'
size 43 first at 1
1: ANYOF(11)
11: EXACT <d>(13)

```

```

13: CURLYX {1,32767}(27)
15: OPEN1(17)
17: EXACT <e>(19)
19: STAR(22)
20: EXACT <f>(0)
22: EXACT <g>(24)
24: CLOSE1(26)
26: WHILEM(0)
27: NOTHING(28)
28: EXACT <h>(30)
30: ANYOF(40)
40: EXACT <k>(42)
42: EOL(43)
43: END(0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
 stclass 'ANYOF' minlen 7

```

La première ligne montre la forme de l'expression avant sa compilation. La seconde sa taille une fois compilée (avec une unité arbitraire, habituellement des mots de 4 octets) et l'*id* du label du premier noeud qui lui correspond.

La dernière ligne (coupée sur deux lignes ci-dessus) contient les infos de l'optimiseur. Dans l'exemple donné, l'optimiseur a trouvé que la correspondance devait contenir une sous-chaîne *de* à l'offset 1, et une sous-chaîne *gh* à un offset quelconque entre 3 et l'infini. Qui plus est, en vérifiant ces sous-chaînes (pour abandonner rapidement les correspondances impossibles) il recherchera la sous-chaîne *gh* avant la sous-chaîne *de*. L'optimiseur peut aussi utiliser le fait qu'il sait que la correspondance doit commencer (au premier *id*) par un caractère, et qu'elle ne doit pas faire moins de 7 caractères.

Les champs intéressants qui peuvent apparaître dans la dernière ligne sont

**anchored** *STRING* at *POS*

**floating** *STRING* at *POS1..POS2*

Voir ci-dessus.

**matching floating/anchored**

Quelle sous-chaîne rechercher en premier.

**minlen**

La longueur minimale de la correspondance.

**stclass** *TYPE*

Type du premier noeud correspondant.

**noscan**

Ne pas rechercher les sous-chaînes trouvées.

**isall**

Indique que les infos de l'optimiseur représentent en fait tout ce que contient l'expression rationnelle, on n'a donc pas du tout besoin d'entrer dans le moteur d'expressions rationnelles.

**GPOS**

Mis si le motif contient \G.

**plus**

Mis si le motif débute par un caractère répété (comme dans *x+y*).

**implicit**

Mis si le motif commence par *.\**.

**with eval**

Mis si le motif contient des groupes *eval*, tels que *(?{ code })* et *(??{ code })*.

**anchored(TYPE)**

Mis si le motif ne peut correspondre qu'à quelques endroits (avec *TYPE* valant *BOL*, *MBOL* ou *GPOS*, voir la table ci-dessous).

Si une sous-chaîne est connue comme ne pouvant correspondre qu'à une fin de ligne, elle peut être suivie de *\$*, comme dans *floating 'k'\$*.

Les infos spécifiques de l'optimiseur sont utilisées pour éviter d'entrer dans un moteur d'expressions rationnelles (lent) pour des chaînes qui ne correspondront certainement pas. Si le drapeau *isall* est mis, un appel du moteur d'expressions rationnelles peut être évité même lorsque l'optimiseur a trouvé un endroit approprié pour la correspondance.

Le reste de la sortie contient la liste des *noeuds* de la forme compilée de l'expression. Chaque ligne a pour format

*id: TYPE OPTIONAL-INFO (next-id)*

### 56.4.2 Types de noeuds

Voici la liste des types possibles accompagnés de courtes descriptions :

```
TYPE arg-description [num-args] [longjump-len] DESCRIPTION

Points de sortie
END no Fin du programme.
SUCCEED no Retour d'un sous-programme, simplement.

Ancres
BOL no Correspond à "" en début de ligne.
MBOL no Idem, sur plusieurs lignes.
SBOL no Idem, sur une seule ligne.
EOS no Correspond à "" en fin de chaîne.
EOL no Correspond à "" en fin de ligne.
MEOL no Idem, sur plusieurs lignes.
SEOL no Idem, sur une seule ligne.
BOUND no Correspond à "" à une frontière entre mots.
BOUNDL no Correspond à "" à une frontière entre mots.
NBOUND no Correspond à "" en-dehors d'une frontière.
NBOUNDL no Correspond à "" en-dehors d'une frontière.
GPOS no Correspondance là où le dernier m//g s'est arrêté.

Alternatives [spéciales]
ANY no Correspond à n'importe quel caractère (sauf
nouvelle ligne)
SANY no Correspond à un caractère.
ANYOF sv Correspond à un caractère dans (ou hors de)
cette classe.
ALNUM no Correspond à un caractère alphanumérique.
ALNUML no Correspond à un caractère alphanumérique local.
NALNUM no Correspond à un caractère non alphanumérique
NALNUML no Correspond à un caractère non alphanumérique local.
SPACE no Correspond à un blanc.
SPACEL no Correspond à un blanc local.
NSPACE no Correspond à un caractère non blanc.
NSPACEL no Correspond à un caractère non blanc local.
DIGIT no Correspond à un caractère numérique.
NDIGIT no Correspond à un caractère non numérique.

BRANCH
L'ensemble de branches constituant un simple choix,
accompagnées de leurs pointeurs "suivant", puisque la
précedence empêche quoi que ce soit d'être concaténé à
une branche particulière. Le pointeur "suivant" de la
dernière BRANCH dans un choix pointe vers ce qui suit
le choix complet. C'est aussi là que pointe le
pointeur "suivant" final de chaque branche ; chaque
branche débute par le noeud opérande d'un noeud BRANCH.
#
BRANCH node Correspond à cette alternative, ou la suivante...

BACK
Les pointeurs "suivant" normaux pointent tous
implicitement vers l'avant ; BACK existe pour rendre
les structures de boucles possibles.
non utilisé
BACK no Correspond à "", le pointeur "suivant" pointe
vers l'arrière.
```

```

Littéraux
EXACT sv Correspond à cette chaîne (précédée de sa
 longueur).
EXACTF sv Correspond à cette chaîne, repliée (? NDT) (avec sa
 longueur).
EXACTFL sv Correspond à cette chaîne locale, repliée
 (avec sa longueur).

Ne fait rien
NOTHING no Correspond à la chaîne vide.
Une variante du précédent, qui délimite un groupe, arrêtant
ainsi les optimisations
TAIL no Correspond à la chaîne vide. On peut sauter
 d'ici vers l'extérieur.

STAR,PLUS '?', et les complexes '*' et '+', sont implémentés
sous la forme de structure BRANCH circulaires
utilisant BACK. Les cas simples (un caractère par
correspondance) sont implémentés avec STAR et PLUS
pour leur rapidité et pour minimiser les plongées
récursives.
#
STAR node Correspond à ce (simple) truc 0 ou plusieurs fois.
PLUS node Correspond à ce (simple) truc 1 ou plusieurs fois.

CURLY sv 2 Correspond à ce simple truc {n,m} fois.
CURLYN no 2 Match next-after-this simple thing
{n,m} times, set parens.
CURLYM no 2 Correspond à ce truc de complexité moyenne {n,m} fois.
CURLYX sv 2 Correspond à ce truc complexe {n,m} fois.

Ce terminateur crée une structure de boucle pour CURLYX
WHILEM no Effectue un traitement des accolades et voit
 si le reste correspond.

OPEN,CLOSE,GROUPP ... sont dénombrés à la compilation.
OPEN num 1 Marque ce point de l'entrée comme début #n.
CLOSE num 1 Analogue à OPEN.

REF num 1 Correspond à une chaîne déjà trouvée.
REFF num 1 Correspond à une chaîne déjà trouvée, repliée.
REFFL num 1 Correspond à une chaîne déjà trouvée, repliée,
 locale.

assertions de groupage
IFMATCH off 1 2 Réussit si la suite correspond.
UNLESSM off 1 2 Rate si la suite correspond.
SUSPEND off 1 1 Sous expression rationnelle "indépendante".
IFTHEN off 1 1 Switch, devrait être précédé par un switcher.
GROUPP num 1 Si le groupe correspond.

Support des expressions rationnelles longues
LONGJMP off 1 1 Saute loin en avant.
BRANCHJ off 1 1 BRANCH avec un offset long.

Le travailleur de force
EVAL evl 1 Exécute du code Perl.

Modifieurs
MINMOD no L'opérateur suivant n'est pas avide.
LOGICAL no L'opcode suivant doit placer le drapeau
 uniquement (? NDT).

```

```
Ceci n'est pas encore utilisé
RENUM off 1 1 Groupe ayant des parenthèses dénombrées
 indépendamment.

Ceci n'est pas vraiment un noeud, mais un morceau optimisé d'un
noeud "long". Pour simplifier la sortie de débogage, nous
l'indiquons comme si c'était un noeud
OPTIMIZED off Conteneur pour vidage.
```

### 56.4.3 Sortie lors de l'exécution

Tout d'abord, lors de la recherche d'une correspondance, on peut n'avoir aucune sortie même si le débogage est validé. Ceci signifie qu'on n'est jamais entré dans le moteur d'expressions rationnelles, et que tout le travail a été fait par l'optimiseur.

Si on est entré dans le moteur d'expressions rationnelles, la sortie peut avoir cette allure :

```
Matching '[bc]d(ef*g)+h[ij]k$' against 'abcdefg__gh__'
Setting an EVAL scope, savestack=3
 2 <ab> <cdefg__gh__> | 1: ANYOF
 3 <abc> <cdefg__gh__> | 11: EXACT <d>
 4 <abcd> <efg__gh__> | 13: CURLYX {1,32767}
 4 <abcd> <efg__gh__> | 26: WHILEM
 0 out of 1..32767 cc=ffff31c
 4 <abcd> <efg__gh__> | 15: OPEN1
 4 <abcd> <efg__gh__> | 17: EXACT <e>
 5 <abcde> <fg__gh__> | 19: STAR
 EXACT <f> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 6 <bcdef> <g__gh__> | 22: EXACT <g>
 7 <bcdefg> <__gh__> | 24: CLOSE1
 7 <bcdefg> <__gh__> | 26: WHILEM
 1 out of 1..32767 cc=ffff31c
Setting an EVAL scope, savestack=12
 7 <bcdefg> <__gh__> | 15: OPEN1
 7 <bcdefg> <__gh__> | 17: EXACT <e>
 restoring \1 to 4(4)..7
 failed, try continuation...
 7 <bcdefg> <__gh__> | 27: NOTHING
 7 <bcdefg> <__gh__> | 28: EXACT <h>
 failed...
 failed...
```

L'information la plus significative de la sortie est celle concernant le *noeud* particulier de l'expression rationnelle compilée qui est en cours de test vis-à-vis de la chaîne cible. Le format de ces lignes est le suivant

*STRING-OFFSET* <PRE-STRING> <POST-STRING> |ID: TYPE

Les infos de *TYPE* sont indentées en fonction du niveau de trace. D'autres informations incidentes apparaissent entremêlées au reste.

## 56.5 Débogage de l'utilisation de la mémoire par Perl

Perl est *très* frivole avec la mémoire. Il y a un dicton qui dit que pour estimer l'utilisation de la mémoire par Perl, il faut envisager un algorithme d'allocation raisonnable et multiplier votre estimation par 10, et même si vous êtes peut-être encore loin du compte, au moins vous ne serez pas trop surpris. Ce n'est pas absolument vrai, mais cela peut vous donner une bonne idée de ce qui se passe.

Disons qu'un entier ne peut pas occuper moins de 20 octets en mémoire, qu'un flottant ne peut pas prendre moins de 24 octets, qu'une chaîne ne peut pas prendre moins de 32 octets (tous ces exemples valant pour des architectures 32 bits, les résultats étant bien pires sur les architectures 64 bits). Si on accède à une variable de deux ou trois façons différentes (ce qui requiert un entier, un flottant ou une chaîne), l'empreinte en mémoire peut encore augmenter de 20 octets. Une implémentation peu soignée de `malloc()` augmentera encore plus ces nombres.

À l'opposé, une déclaration comme

```
sub foo;
```

peut prendre jusqu'à 500 octets de mémoire, selon la version de Perl que vous utilisez.

Des estimations à la louche et anecdotiques sur un code bouffi donnent un facteur d'accroissement d'environ 8. Cela signifie que la forme compilée d'un code raisonnable (commenté normalement, indenté proprement, etc.) prendra approximativement 8 fois plus de place que l'espace disque nécessaire au code.

Il existe deux façons spécifiques à Perl d'analyser l'usage de la mémoire : `$ENV{PERL_DEBUG_MSTATS}` et l'option de ligne de commande `-DL`. La première est disponible seulement si perl est compilé avec le `malloc()` de Perl, la seconde seulement si Perl a été compilé avec l'option `-DDEBUGGING`. Voir les instructions sur la façon dont on fait cela dans la page pod `INSTALL` à la racine de l'arborescence des sources de Perl.

### 56.5.1 Utilisation de `$ENV{PERL_DEBUG_MSTATS}`

Si votre perl utilise le `malloc()` de Perl, et s'il a été compilé avec les options correctes (c'est le cas par défaut), alors il affichera des statistiques sur l'usage de la mémoire après avoir compilé votre code lorsque `$ENV{PERL_DEBUG_MSTATS} > 1`, et avant la fin du programme lorsque `$ENV{PERL_DEBUG_MSTATS} >= 1`. Le format du rapport est similaire à celui de l'exemple suivant :

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
 14216 free: 130 117 28 7 9 0 2 2 1 0 0
 437 61 36 0 5
 60924 used: 125 137 161 55 7 8 6 16 2 0 1
 74 109 304 84 20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution: (buckets 4(4)..8188(8192)
 30888 free: 245 78 85 13 6 2 1 3 2 0 1
 315 162 39 42 11
175816 used: 265 176 1112 111 26 22 11 27 2 1 1
 196 178 1066 798 39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail: 0+2192+0+6144.
```

Il est possible de demander de telles statistiques à un moment arbitraire de votre exécution en utilisant la fonction `mstats()` du module standard `Devel::Peek::mstats()`.

Voici l'explication des différentes parties du format :

#### **buckets SMALLEST(APPROX) .. GREATEST(APPROX)**

Le `malloc()` de Perl utilise des allocations par buckets. Chaque requête est arrondie à la plus proche taille de bucket disponible, et un bucket de cette taille est pris dans le pool de buckets correspondant.

La ligne ci-dessus décrit les limites des buckets en cours d'utilisation. Chaque bucket a deux tailles : l'empreinte en mémoire, et la taille maximale des données utilisateur qui peuvent être placées dans ce bucket. Supposez dans l'exemple ci-dessus que la taille du bucket le plus petit est de 4. Le plus grand bucket aura une taille utilisable de 8188, et son empreinte en mémoire sera de 8192.

#### **Free/Used**

La rangée ou les deux rangées suivante(s) de nombres correspond(ent) au nombre de buckets de chaque taille entre `SMALLEST` et `GREATEST`. Dans la première rangée, les tailles (empreintes mémoire) des buckets sont des puissances de deux (ou peut-être d'une page plus grandes). Dans la seconde rangée (si elle est présente) les empreintes en mémoire des buckets sont entre les empreintes mémoire des deux buckets de la rangée au-dessus.

Par exemple, supposons dans l'exemple précédent que les empreintes mémoire soient de

```
free: 8 16 32 64 128 256 512 1024 2048 4096 8192
 4 12 24 48 80
```

Sans `DÉBOGAGE` de perl les buckets ayant une longueur supérieure à 128 ont un en-tête de 4 octets, un bucket de 8192 octets de long peut ainsi supporter des allocations de 8188 octets.

#### **Total sbrk() : SBRKed/SBRKs:CONTINUOUS**

Les deux premiers champs donnent la quantité totale de mémoire que perl a `sbrk()`é (ess-broken? :-), et le nombre de `sbrk()`s utilisés. Le troisième nombre est ce que perl pense de la continuité des morceaux retournés. Tant que ce nombre est positif, `malloc()` présumera qu'il est probable que `sbrk()` fournira une mémoire continue.

La mémoire allouée par les bibliothèques externes n'est pas comptée.

**pad: 0**

La quantité de mémoire sbrk()ée nécessaire pour garder les buckets alignés.

**heads: 2192**

Tandis que l'en-tête en mémoire des buckets les plus grands est gardée à l'intérieur du bucket, pour les buckets plus petits, il est stocké dans des zones séparées. Ce champ donne la taille totale de ces zones.

**chain: 0**

malloc() peut vouloir diviser un gros bucket en buckets plus petits. Si seulement une part du bucket décédé est laissée non subdivisée, le reste gardé comme élément d'une liste chaînée. Ce champ donne la taille totale de ces morceaux.

**tail: 6144**

Pour minimiser la quantité de sbrk()s malloc() demande plus de mémoire. Ce champ donne la taille de la partie non encore utilisée, qui est sbrk()ée, mais jamais touchée.

## 56.5.2 Exemple d'utilisation de l'option -DL

Ci-dessous nous montrons comment analyser l'usage de la mémoire par

```
do 'lib/auto/POSIX/autosplit.ix';
```

Le fichier en question contient un en-tête et 146 lignes similaires à

```
sub getcwd;
```

**AVERTISSEMENT** : la discussion ci-dessous suppose une architecture 32 bits. Dans les version de perl les plus récentes, l'usage de la mémoire des constructions discutées ici est nettement améliorée, mais ce qui suit est une histoire vraie. Cette histoire est impitoyablement laconique, et suppose un peu plus qu'une connaissance superficielle du fonctionnement interne de Perl. Appuyez sur espace pour continuer, 'q' pour quitter (en fait, vous voudrez juste passer à la section suivante).

Voici la liste détaillée des allocations réalisées par Perl pendant l'analyse de ce fichier :

```
!!! "after" at test.pl line 3.
 Id subtot 4 8 12 16 20 24 28 32 36 40 48 56 64 72 80 80+
0 02 13752 294 4
0 54 5545 . . 8 124 16 . . . 1 1 3
5 05 32 1
6 02 7152 149
7 02 3600 150
7 03 64 . -1 . 1 . . 2
7 04 7056 7
7 17 38404 1 . . 442 149 . . 147 . .
9 03 2078 17 249 32 2
```

Pour voir cette liste, insérez deux instructions warn('!...') autour de l'appel :

```
warn('!');
do 'lib/auto/POSIX/autosplit.ix';
warn('!!! "after"');
```

et exécutez-le avec l'option **-DL**. Le premier warn() affichera les infos sur l'allocation mémoire avant l'analyse du fichier, et mémorisera les statistiques à cet instant (nous ignorons ce qu'il affiche). Le second warn() affichera les variations par rapport à ces statistiques. Cela donne la sortie précédente.

Les différents *Id*entifiants sur la gauche correspondent aux différents sous-systèmes de l'interpréteur perl, ils sont juste les premiers arguments donnés à l'API d'allocation mémoire New() de perl. Pour déterminer ce que 9 03 signifie faites un grep dans le source de perl à la recherche de 903. Vous verrez que c'est la fonction savepn() dans *util.c*. Cette fonction est utilisée pour stocker une copie d'un morceau existant de mémoire. En utilisant un débogueur C, on peut voir qu'elle est appelée soit directement depuis gv\_init(), ou via sv\_magic(), et gv\_init() est appelée depuis gv\_fetchpv() - qui est appelée depuis newSUB(). S'il-vous-plaît, veuillez faire une pause pour reprendre votre souffle maintenant.



**NOTE** : pour atteindre cet endroit dans le débogueur et sauter tous les appels à `savepv` pendant la compilation du script principal, placez un point d'arrêt C dans `Perl_warn()`, continuez jusqu'à ce que ce point soit atteint, puis placez un point d'arrêt dans `Perl_savepv()`. Notez que vous pouvez avoir besoin de sauter une poignée de `Perl_savepv()` qui ne correspondent pas à une production de masse de CV (il y a plus d'allocations 903 que les 146 lignes identiques de *lib/auto/POSIX/autosplit.ix*). Notez aussi que les préfixes `Perl_` sont ajoutés par du code de macroisation dans les fichiers d'en-tête perl pour éviter des conflits avec les bibliothèques externes.

En tout cas, nous voyons que les ids 903 correspondent à la création de globs, deux fois par glob - pour le nom du glob, et pour la magie de transformation en chaîne du glob (Wouarf ! NDT).

Voici des explications pour les autres *Ids* ci-dessus :

#### 717

Crée de plus grosses structures `XPV*`. Dans le cas ci-dessus, il crée 3 AV par sous-programme, un pour une liste de noms de variables lexicales, un pour un scratchpad (qui contient les variables lexicales et les cibles), et un pour le tableau des scratchpads nécessaire pour la récursivité.

Il crée aussi un GV et un CV par sous-programme, tous appelés depuis `start_subparse()`.

#### 002

Crée un tableau C correspondant à l'AV des bloc-notes (NDT ?), et le bloc-note lui-même. La première entrée fautive de ce bloc-note est créée même si le sous-programme lui-même n'est pas encore défini.

Il crée aussi des tableaux C pour conserver les données mises de côté (stash NDT ?) (c'est un HV (NDT ?), mais il grossit, il se produit donc 4 grosses allocations : les gros paquets ne sont pas libérés, et sont conservés comme arènes (NDT ?) additionnelles pour les allocations de SV).

#### 054

Crée un HEK pour le nom du glob (NDT ?) du sous-programme (ce nom est une clé dans un *stash*).

Les grosses allocations ayant cet *Id* correspondent à des allocations de nouvelles arènes pour stocker HE.

#### 602

Crée un GP pour le glob du sous-programme.

#### 702

Crée le MAGIC pour le glob du sous-programme.

#### 704

Crée des arènes qui stockent les SV.

### 56.5.3 Détails sur -DL

Si Perl est exécuté avec l'option **-DL**, alors les `warn()`s qui débutent par `!` se comportent de façon particulière. Ils affichent une liste des *catégories* d'allocations de mémoire, et des statistiques sur leurs tailles.

Si la chaîne `warn()` commence par

!!!

affiche uniquement les catégories ayant changé, affiche les variations en nombre d'allocations.

!!

affiche uniquement les catégories ayant grandi, leurs nombres en valeur absolue, et leurs totaux.

!

affiche les catégories non vides, leurs nombres en valeur absolue, et leurs totaux.

### 56.5.4 Limitations des statistiques -DL

Si une extension ou une bibliothèque externe n'utilise pas l'API Perl pour attribuer la mémoire, de telles allocations ne sont pas comptées.

## 56.6 VOIR AUSSI

*perldebug*, *perlguts*, *perlrun re*, et *Devel::Dprof*.

## **56.7 TRADUCTION**

### **56.7.1 Version**

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### **56.7.2 Traducteur**

Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>

### **56.7.3 Relecture**

G rard Delafond

# Chapitre 57

## perlXStut

Guide d'apprentissage des XSUB

### 57.1 DESCRIPTION

Ce guide fournit au lecteur les étapes à suivre pour réaliser une extension de Perl. On suppose qu'il a accès à *perlguts* et à *perlx*.

Le guide commence par des exemples très simples, avant d'aborder des points plus complexes, chaque nouvel exemple introduisant des fonctionnalités supplémentaires. Certains concepts ne seront pas expliqués complètement du premier coup, afin de faciliter au lecteur l'apprentissage progressif de la construction d'extensions.

#### 57.1.1 VERSION DE CE DOCUMENT

Nous nous efforçons de maintenir ce guide à jour par rapport aux dernières versions de développement de Perl. Cela signifie qu'il est parfois en avance sur la dernière version stable de Perl, et qu'il se peut que des fonctionnalités décrites ici soient absentes dans les versions plus anciennes. Cette section conserve la trace des fonctionnalités ajoutées à Perl 5.

– Dans les versions de Perl 5.002 antérieures à la version gamma, le script de test de l'exemple 1 ne fonctionnera pas correctement. Vous devez modifier la ligne "use lib" comme suit :

```
use lib './b1ib';
```

– Dans les versions de Perl 5.002 antérieures à la version beta 3, la ligne du fichier .xs contenant "PROTYPES: DISABLE" entraînera une erreur de compilation. Supprimez simplement cette ligne.

– Dans les versions de Perl 5.002 antérieures à la version 5.002b1h, le fichier test.pl n'était pas créé automatiquement par h2xs. Cela signifie que vous ne pouvez pas exécuter "make test" pour lancer le script de test. Vous devrez rajouter la ligne suivante avant l'instruction "use extension" :

```
use lib './b1ib';
```

– Dans les versions 5.000 et 5.001, vous ne devez pas utiliser la ligne qui précède, mais plutôt celle-ci :

```
BEGIN { unshift(@INC, './b1ib') }
```

– On suppose dans ce document que l'exécutable dont le nom est "perl" correspond à Perl, version 5. Dans certains systèmes, Perl 5 peut avoir été installé sous le nom "perl5".

#### 57.1.2 DYNAMIQUE / STATIQUE

On croit souvent que, si un système ne supporte pas les bibliothèques dynamiques, il est impossible d'y construire des XSUB. C'est une erreur. Vous *pouvez* les construire, mais, à l'édition de liens, vous devez assembler les sous-routines de la XSUB au reste de Perl à l'intérieur d'un nouvel exécutable. La situation est la même que dans Perl 4.

Ce guide peut quand même être utilisé sur un tel système. Le système de compilation de la XSUB détectera le système et construira, si cela est possible, une bibliothèque dynamique ; sinon, il construira une bibliothèque statique accompagnée, de manière optionnelle, d'un nouvel exécutable contenant cette bibliothèque liée de manière statique.

Supposons que vous souhaitiez construire un exécutable lié statiquement, sur un système supportant les bibliothèques dynamiques. Dans ce cas, chaque fois que la commande "make" sans argument est exécutée dans les exemples qui suivent, vous devrez utiliser la commande "make perl" à la place.

Si vous avez choisi de générer un tel exécutable lié statiquement, vous devrez aussi remplacer "make test" par "make test\_static". Sur les systèmes qui ne peuvent pas du tout construire de bibliothèque dynamique, "make test" est suffisant.

### 57.1.3 EXAMPLE 1

Notre première extension sera très simple. Lorsque nous appellerons la routine définie dans l'extension, elle affichera un message donné et se terminera.

Exécutez `h2xs -A -n Montest`. Cela crée un répertoire nommé Montest, éventuellement sous `ext/` si ce répertoire existe dans le répertoire courant. Plusieurs fichiers seront créés sous `Montest/`, en particulier `MANIFEST`, `Makefile.PL`, `Montest.pm`, `Montest.xs`, `test.pl` et `Changes`.

Le fichier `MANIFEST` contient les noms de tous les fichiers créés.

Le fichier `Makefile.PL` devrait ressembler à ceci :

```
use ExtUtils::MakeMaker;
See lib/ExtUtils/MakeMaker.pm for details of how to influence
the contents of the Makefile that is written.
WriteMakefile(
 'NAME' => 'Montest',
 'VERSION_FROM' => 'Montest.pm', # finds $VERSION
 'LIBS' => [''], # e.g., '-lm'
 'DEFINE' => '', # e.g., '-DHAVE_SOMETHING'
 'INC' => '', # e.g., '-I/usr/include/other'
);
```

Le fichier `Montest.pm` devrait commencer à peu près comme suit :

```
package Montest;

require Exporter;
require DynaLoader;

@ISA = qw(Exporter DynaLoader);
Items to export into callers namespace by default. Note: do not export
names by default without a very good reason. Use EXPORT_OK instead.
Do not simply export all your public functions/methods/constants.
@EXPORT = qw(

);
$VERSION = '0.01';

bootstrap Montest $VERSION;

Preloaded methods go here.

Autoload methods go after =cut, and are processed by the autosplit program.

1;
__END__
Below is the stub of documentation for your module. You better edit it!
```

Et voici à quoi devrait ressembler le fichier `Montest.xs` :

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif
```

```
PROTOTYPES: DISABLE
```

```
MODULE = Montest
```

```
PACKAGE = Montest
```

Modifions le fichier .xs en ajoutant ceci à la fin :

```
void
bonjour()
 CODE:
 printf("Bonjour !\n");
```

A présent, lançons "perl Makefile.PL", ce qui va créer un vrai Makefile, nécessaire pour make. Les résultats suivants seront affichés :

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Montest
%
```

En lançant maintenant make, nous obtenons à peu près la sortie suivants (certaines lignes sont tronquées pour plus de clarté) :

```
% make
umask 0 && cp Montest.pm ./blib/Montest.pm
perl xsubpp -typemap typemap Montest.xs >Montest.tc && mv Montest.tc Montest.c
cc -c Montest.c
Running Mkbootstrap for Montest ()
chmod 644 Montest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Montest/Montest.sl -b Montest.o
chmod 755 ./blib/PA-RISC1.1/auto/Montest/Montest.sl
cp Montest.bs ./blib/PA-RISC1.1/auto/Montest/Montest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Montest/Montest.bs
```

A présent, bien qu'il y ait déjà un patron test.pl prêt à l'emploi, nous allons, pour cet exemple seulement, réaliser un script de test spécial. Créez un fichier avec le nom bonjour, qui contiendra ce qui suit :

```
#!/opt/perl5/bin/perl

use ExtUtils::testlib;

use Montest;

Montest::bonjour();
```

A ce point, en lançant le script, vous devriez obtenir la sortie suivante :

```
% perl bonjour
Bonjour !
%
```

### 57.1.4 EXAMPLE 2

Nous allons maintenant ajouter dans notre extension une sous-routine qui prendra un argument et renverra 1 si l'argument est pair, 0 s'il est impair.

Ajoutez les lignes suivantes à la fin de Montest.xs :

```
int
est_pair(entree)
 int entree
 CODE:
 RETVAL = (entree % 2 == 0);
 OUTPUT:
 RETVAL
```

Les espaces blancs au début de la ligne "int entree" ne sont pas obligatoires, mais ils améliorent la lisibilité. Le point-virgule à la fin de la même ligne est aussi facultatif.

"int" et "entree" peuvent être séparés par des blancs. On pourrait aussi se passer d'indenter les quatre lignes à partir de celle qui commence par "CODE:". Mais il est recommandé, pour des raisons de lisibilité, d'utiliser une indentation de 8 espaces (ou une tabulation normale).

A présent, relancez make pour reconstruire la librairie partagée.

Puis suivez à nouveau les étapes ci-dessus pour générer un Makefile à partir du fichier Makefile.PL et lancer make.

Pour vérifier que l'extension fonctionne, nous devons regarder le fichier test.pl. Ce fichier est organisé de manière à simuler le système de tests de Perl lui-même. Dans ce script, vous lancez une série de tests qui vérifient le comportement de l'extension, en affichant "ok" lorsque le test réussit, "not ok" dans le cas contraire. Modifiez l'instruction print dans le bloc BEGIN pour afficher "1..4", et ajoutez le code suivant à la fin du fichier :

```
print &Montest::est_pair(0) == 1 ? "ok 2" : "not ok 2", "\n";
print &Montest::est_pair(1) == 0 ? "ok 3" : "not ok 3", "\n";
print &Montest::est_pair(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

Nous allons appeler le script de test avec la commande "make test". Vous devriez obtenir une sortie comme la suivante :

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.002b2/bin/perl (nombreux arguments -I) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

### 57.1.5 QUE S'EST-IL DONC PASSE ?

Le programme h2xs est le point de départ de la création d'extensions. Dans les exemples à venir, nous verrons comment utiliser h2xs pour lire des fichiers d'en-tête et générer des patrons pour se connecter à des routines en C.

h2xs crée plusieurs fichiers dans le répertoire de l'extension. Le fichier Makefile.PL est un script Perl qui va générer un vrai Makefile pour construire l'extension. Nous verrons ce point plus en détail dans un moment.

Les fichiers <extension>.pm et <extension>.xs contiennent le coeur de l'extension. Dans le fichier .xs se trouvent les routines en C qui constitueront l'extension. Le fichier .pm contient des routines qui indiqueront à Perl comment charger l'extension.

En générant puis en invoquant le Makefile, on crée un répertoire blib (c'est-à-dire "binary library" ou librairie binaire) dans le répertoire courant. Ce répertoire contiendra la librairie partagée que nous construirons. Après l'avoir testée, nous pourrions installer celle-ci à son emplacement final.

L'utilisation de "make test" pour lancer le programme de test fait quelque chose de très important : cette commande appelle Perl avec tous les arguments -I, ce qui lui permet de trouver les divers fichiers constituant l'extension.

Il est *primordial* d'utiliser "make test" tant que vous êtes encore en train de tester les extensions. Si vous essayez de lancer le script de test directement, vous aurez une erreur fatale.

Une autre raison de l'importance de passer par "make test" pour lancer votre script de test est que, lorsque vous testez la mise à jour d'une version existante, "make test" garantit que vous utilisez votre nouvelle extension et non la version existante.

Lorsque Perl voit un `use extension;`, il cherche un fichier qui ait le même nom que l'extension faisant l'objet du `use`, avec un suffixe `.pm`. S'il ne peut pas trouver ce fichier, Perl s'arrête avec une erreur fatale. Le chemin de recherche par défaut est contenu dans le tableau `@INC`.

Dans notre cas, `Montest.pm` indique à Perl qu'il aura besoin des extensions `Exporter` et `Dynamic Loader`. Puis il remplit les tableaux `@ISA` et `@EXPORT`, ainsi que le scalaire `$VERSION`; finalement, il demande à Perl d'initialiser l'extension. Perl appellera la routine de chargement dynamique (si elle existe) et chargera la librairie dynamique.

Les deux tableaux qui sont positionnés dans le fichier `.pm` ont une importance particulière. `@ISA` contient une liste de paquetages où doivent être recherchées les méthodes (ou les sous-routines) qui n'existent pas dans le paquetage courant. Le tableau `@EXPORT` indique à Perl quelles routines, parmi celles de l'extension, doivent être placées dans l'espace de nommage du paquetage appelant.

Il faut vraiment choisir avec soin ce qu'on exporte. N'exportez JAMAIS des noms de méthodes, et n'exportez PAS autre chose *par défaut* sans une bonne raison.

En règle générale, si le module est orienté objet, n'exportez rien du tout. S'il s'agit seulement d'un ensemble de fonctions, vous pouvez alors exporter n'importe quelle fonction dans un autre tableau, `@EXPORT_OK`.

Consultez *perlmod* pour plus d'informations.

La variable `$VERSION` a pour rôle de garantir que le fichier `.pm` et la librairie partagée sont "en phase" l'un avec l'autre. Chaque fois que vous modifiez le fichier `.pm` ou `.xs`, vous devriez incrémenter la valeur de cette variable.

### 57.1.6 ECRIRE DE BONS SCRIPTS DE TEST

On ne dira jamais trop combien il est important de rédiger de bons programmes de test. Vous devriez suivre de près le style "ok/not ok" utilisé par Perl lui-même, afin que chaque test puisse être interprété très facilement et sans ambiguïté. Lorsque vous trouvez un bug et que vous le corrigez, n'oubliez pas de rajouter un test unitaire à son sujet.

En lançant "make test", vous vous assurez que le script `test.pl` fonctionne et qu'il utilise votre extension dans la bonne version. Si vous avez un grand nombre de tests unitaires, peut-être voudrez-vous imiter la structure des fichiers de tests de Perl. Créez un répertoire nommé "t", et donnez à tous vos scripts de test la terminaison ".t". Le Makefile lancera alors tous ces tests de la manière adéquate.

### 57.1.7 EXEMPLE 3

Notre troisième extension prendra un argument en entrée, arrondira sa valeur et la remettra dans l'*argument* lui-même.

Rajoutez le code qui suit à la fin de `Montest.xs` :

```
void
arrondir(arg)
 double arg
 CODE:
 if (arg > 0.0) {
 arg = floor(arg + 0.5);
 } else if (arg < 0.0) {
 arg = ceil(arg - 0.5);
 } else {
 arg = 0.0;
 }
 OUTPUT:
 arg
```

Modifiez le fichier `Makefile.PL` de manière à ce que la ligne "LIBS" ressemble à ce qui suit :

```
'LIBS' => ['-lm'], # e.g., '-lm'
```

Générez le Makefile, et lancez `make`. Dans `test.pl`, modifiez le bloc `BEGIN` afin d'afficher "1..9", et ajoutez ceci :

```

$i = -1.5; &Montest::arrondir($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Montest::arrondir($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Montest::arrondir($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Montest::arrondir($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Montest::arrondir($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";

```

"make test" devrait à présent dire que les neuf tests sont réussis.

Vous vous demandez peut-être s'il est possible d'arrondir une constante. Pour voir ce qui se passe, ajoutez temporairement la ligne suivante dans test.pl :

```
&Montest::arrondir(3);
```

Lancez "make test", et remarquez que Perl s'arrête avec une erreur fatale. Perl ne vous laisse pas modifier la valeur des constantes !

### 57.1.8 QU'Y A-T-IL DE NOUVEAU ICI ?

Il y a deux nouveautés ici. D'abord, nous avons apporté des modifications à Makefile.PL. En l'occurrence, nous avons spécifié une librairie supplémentaire à lier dans l'exécutable, la librairie mathématique libm. Nous expliquerons plus loin comment écrire des XSUB qui appellent une routine quelconque dans une librairie.

Ensuite, le résultat est renvoyé à l'appelant non par la valeur de retour de la fonction, mais par la variable qui a été passée à la fonction comme argument.

### 57.1.9 PARAMETRES D'ENTREE ET DE SORTIE

Vous précisez les paramètres à passer à la XSUB juste après avoir déclaré la valeur de retour et le nom de la fonction. Les blancs en début de ligne et le point-virgule à la fin sont facultatifs.

La liste des paramètres de sortie est spécifiée après la directive OUTPUT:. L'utilisation de RETVAL indique à Perl que cette valeur doit être renvoyée par la fonction XSUB. Dans l'exemple 3, comme la valeur que nous voulions renvoyer était contenue dans la variable passée en argument elle-même, nous avons rajouté cette variable (et non RETVAL) dans la section OUTPUT:.

### 57.1.10 LE COMPILATEUR XSUBPP

Le compilateur xsubpp prend le code XS dans le fichier .xs et le convertit en code C, en le mettant dans un fichier avec la terminaison .c. Le code C généré fait un grand usage des fonctions C internes à Perl.

### 57.1.11 LE FICHIER TYPEMAP

La compilateur xsubpp se base sur un ensemble de règles pour convertir les types de données Perl (scalaire, tableau, etc...) en types de données C (int, char \*, etc...). Ces règles sont stockées dans le fichier typemap (\$PERLLIB/ExtUtils/typemap), qui est divisé en trois parties.

La première partie tente de rapporter les différents types de données C à un code qui a une relation de correspondance avec les types Perl. La deuxième partie contient du code C utilisé par xsubpp pour les paramètres d'entrée. La troisième partie contient du code C utilisé par xsubpp pour les paramètres de sortie. Nous parlerons en détail du code C plus loin.

Considérons à présent une partie du fichier .c créé pour notre extension.

```

XS(XS_Montest_arrondir)
{
 dXSARGS;
 if (items != 1)
 croak("Usage: Montest::arrondir(arg)");
 {
 double arg = (double)SvNV(ST(0)); /* XXXXX */
 if (arg > 0.0) {
 arg = floor(arg + 0.5);
 } else if (arg < 0.0) {

```



```

 arg = ceil(arg - 0.5);
 } else {
 arg = 0.0;
 }
 sv_setnv(ST(0), (double)arg); /* XXXXX */
}
XSRETURN(1);
}

```

Remarquez les deux lignes marquées d'un "XXXXX". En consultant la première section du fichier `typemap`, vous constaterez que les doubles sont associés au type `T_DOUBLE`. Dans la section `INPUT`, un argument de type `T_DOUBLE` est assigné à une variable en lui appliquant la routine `SvNV`, puis en convertissant le résultat en double, et en l'assignant à la variable. De même, d'après la section `OUTPUT`, utilisée lorsque `arg` possède sa valeur finale, `arg` est passé à la fonction `sv_setnv` pour être renvoyé à la sous-routine appelante. Ces deux fonctions sont décrites dans *perlguts*; nous préciserons plus loin, dans la section concernant la pile des arguments, ce que signifie ce "ST(0)".

### 57.1.12 AVERTISSEMENT

En général, c'est une mauvaise idée d'écrire des extensions qui modifient leurs paramètres d'entrée, comme dans l'exemple 3. Toutefois ce comportement est toléré pour permettre d'appeler de manière plus commode des routines C pré-existantes, lesquelles modifient fréquemment leurs paramètres d'entrée. L'exemple qui suit montre comment faire cela.

### 57.1.13 EXEMPLE 4

Dans cet exemple, nous commencerons à écrire des `XSUB` qui interagissent avec des bibliothèques C prédéfinies. Construisons tout d'abord une petite bibliothèque à nous, et laissons `h2xs` écrire à notre place les fichiers `.pm` et `.xs`.

Créez un nouveau répertoire nommé `Montest2` au même niveau que le répertoire `Montest`. Sous `Montest2`, créez un autre répertoire nommé `malib`, et positionnez-vous dedans.

Nous allons créer à cet endroit quelques fichiers qui généreront une bibliothèque de test. Parmi eux se trouvera un fichier-source en C et un fichier d'en-tête. Nous allons aussi créer un `Makefile.PL` dans ce répertoire. Ensuite nous nous assurerons que l'exécution de `make` au niveau `Montest2` lance automatiquement l'exécution de ce fichier `Makefile.PL` et du `Makefile` résultant.

Dans le répertoire `malib`, créez un fichier `malib.h` ressemblant à ceci :

```

#define TESTVAL 4

extern double toto(int, long, const char*);

```

Créez aussi un fichier `malib.c` avec le contenu suivant :

```

#include <stdlib.h>
#include "./malib.h"

double
toto(a, b, c)
int a;
long b;
const char * c;
{
 return (a + b + atof(c) + TESTVAL);
}

```

Et créez enfin un fichier `Makefile.PL` avec ceci :

```

use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
 NAME => 'Montest2::malib',
 SKIP => [qw(all static static_lib dynamic dynamic_lib)],
 clean => {'FILES' => 'libmalib$(LIB_EXT)'},
);

```

```

sub MY::top_targets {
,
all :: static

static :: libmalib$(LIB_EXT)

libmalib$(LIB_EXT): $(O_FILES)
 $(AR) cr libmalib$(LIB_EXT) $(O_FILES)
 $(RANLIB) libmalib$(LIB_EXT)

,;
}

```

Nous allons maintenant créer les fichiers du répertoire principal Montest2. Allez dans le répertoire au-dessus de Montest2 et lancez la commande suivante :

```
% h2xs -O -n Montest2 ./Montest2/malib/malib.h
```

Vous recevrez un avertissement signalant que Montest2 est écrasé, mais c'est normal. Nos fichiers sont entreposés dans Montest2/malib, et ils ne seront pas modifiés.

Le Makefile.PL normal généré par h2xs ne connaît pas le répertoire malib. Nous devons lui dire qu'il y a un sous-répertoire dans lequel nous allons générer une librairie. Rajoutons le couple clé-valeur suivant dans l'appel de WriteMakefile :

```
'MYEXTLIB' => 'malib/libmalib$(LIB_EXT)',
```

ainsi qu'une nouvelle sous-routine de remplacement :

```

sub MY::postamble {
,
$(MYEXTLIB): malib/Makefile
 cd malib && $(MAKE) $(PASTHRU)
,;
}

```

(Remarque : La plupart des make imposent que la ligne `cd malib && $(MAKE) $(PASTHRU)` soit indentée avec une tabulation, de même que pour le Makefile dans le sous-répertoire)

Adaptons aussi le fichier MANIFEST afin qu'il reflète correctement le contenu de notre extension. La ligne spécifiant "malib" devrait être remplacée par les trois lignes suivantes :

```

malib/Makefile.PL
malib/malib.c
malib/malib.h

```

Afin de conserver un espace de nommage cohérent et non pollué, ouvrez le fichier .pm et modifiez les lignes initialisant @EXPORT et @EXPORT\_OK (il y en a deux : une sur la ligne commençant par "use vars" et l'autre lors de l'initialisation elle-même du tableau). Enfin, dans le fichier .xs, modifiez ainsi la ligne #include :

```
#include "malib/malib.h"
```

Et rajoutez aussi cette définition de fonction à la fin du fichier .xs :

```

double
toto(a,b,c)
 int a
 long b
 const char * c
 OUTPUT:
 RETVAL

```

Maintenant nous devons aussi créer un fichier typemap, car Perl, par défaut, ne supporte pas actuellement le type `const char *`. Créez un fichier nommé `typemap` et mettez-y la ligne suivante :

```
const char * T_PV
```

A présent, lancez Perl sur le `Makefile.PL` au niveau le plus élevé. Remarquez qu'il crée aussi un `Makefile` dans le répertoire `malib`. Lancez `make`, et constatez qu'il va bien dans le répertoire `malib` pour y faire là aussi un `make`.

A présent, éditez le script `test.pl` et modifiez le block `BEGIN` pour afficher "1.4", et ajoutez les lignes suivantes à la fin du script :

```
print &Montest2::toto(1, 2, "Bonjour !") == 7 ? "ok 2\n" : "not ok 2\n";
print &Montest2::toto(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Montest2::toto(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

(Lorsqu'on fait des comparaisons en virgule flottante, il est souvent bon de ne pas vérifier l'égalité stricte, mais plutôt la différence en-dessous d'un certain facteur epsilon, ici 0,01).

Lancez "make test" et tout devrait être au point.

#### 57.1.14 QUE S'EST-IL PASSE ICI ?

Contrairement aux exemples précédents, nous avons lancé `h2xs` sur un vrai fichier d'en-tête, ce qui a fait apparaître des éléments supplémentaires dans les fichiers `.pm` et `.xs`.

- Dans le fichier `.xs`, il y a maintenant une déclaration `#include`, avec le chemin complet du fichier d'en-tête `malib.h`.
- Un peu de code C a été rajouté dans le fichier `.xs`. La routine `constant` sert à rendre disponibles dans le script Perl (en l'occurrence, en appelant `&main::TESTVAL`) les valeurs définies par `#define` dans le fichier d'en-tête. Il y a aussi du code XS pour permettre l'appel de la routine `constant`.
- Le fichier `.pm` a exporté le nom `TESTVAL` dans le tableau `@EXPORT`. Cela pourrait entraîner des conflits de nommage. Un bon principe est que, si un identifiant spécifié par un `#define` ne doit être utilisé que par les routines C elles-mêmes, et non par l'utilisateur, alors il convient de l'enlever du tableau `@EXPORT`. D'autre part, si cela ne vous gêne pas d'utiliser le nom complet de la variable, vous pouvez supprimer la plupart ou la totalité des éléments du tableau `@EXPORT`.
- Si notre fichier d'en-tête contenait des directives `#include`, elles ne seraient pas prises en compte par `h2xs`. Il n'y a pas de solution satisfaisante pour l'instant.

Nous avons aussi indiqué à Perl la librairie construite dans le sous-répertoire `malib`. Il a suffi pour cela de rajouter la variable `MYEXTLIB` lors de l'appel de `WriteMakefile` et de réécrire la sous-routine `postamble` afin de lui faire exécuter `make` depuis le sous-répertoire. Le `Makefile.PL` associé à la librairie est un peu plus compliqué, mais sans excès. Nous avons là aussi remplacé la sous-routine `postamble` pour y insérer notre propre code. Ce code disait simplement que la librairie à créer devait être une archive statique (et non une librairie dynamique), et fournissait les commandes permettant de la construire.

#### 57.1.15 LE PASSAGE D'ARGUMENTS A XSUBPP

Suite à l'exemple 4, il nous est désormais facile de simuler en Perl des librairies existantes dont les interfaces ne sont pas toujours de conception très rigoureuse. Poursuivons à présent par une discussion des arguments passés au compilateur `xsubpp`.

Lorsque vous spécifiez les arguments dans le fichier `.xs`, vous passez en réalité trois informations pour chacun d'entre eux. La première information est le rang de l'argument dans la liste (premier, second, etc...). La seconde est le type de l'argument ; il s'agit de sa déclaration de type (par exemple `int`, `char*`, etc...). La troisième information donne le mode de passage de l'argument lorsque la fonction de la librairie est appelée par cette `XSUB`. Il faut préciser si un "&" doit être placé devant l'argument, ce qui signifie que l'argument doit être passé comme une adresse sur le type de donnée spécifié.

Il y a une différence entre les deux arguments dans la fonction imaginaire suivante :

```
int
toto(a,b)
 char &a
 char * b
```

Le premier argument de cette fonction serait traité comme un char, il serait assigné à la variable a, et son adresse serait passée à la fonction toto. Le second argument serait traité comme un pointeur vers une chaîne de caractères, et assigné à la variable b. La valeur de b serait passée à la fonction toto. L'appel de la fonction toto qui serait effectivement généré par xsubpp ressemblerait à ceci :

```
toto(&a, b);
```

xsubpp analysera de la même manière les listes d'arguments de fonction suivantes :

```
char &a
char&a
char & a
```

Toutefois, pour faciliter la compréhension, on conseille de placer un "&" près du nom de la variable et loin du type, et de placer un "\*" près du type et loin du nom (comme c'est le cas dans l'exemple ci-dessus). Avec cette technique, il est facile de comprendre exactement ce qui sera passé à la fonction C – ce sera l'expression, quelle qu'elle soit, qui se trouve dans la "dernière colonne".

Vous devriez vraiment vous efforcer de passer à la fonction le type de variable qu'elle attend, lorsque c'est possible. Cela vous évitera de nombreux problèmes à long terme.

### 57.1.16 LA PILE DES ARGUMENTS

En regardant n'importe quel code généré par chacun des exemples, sauf le premier, vous remarquerez plusieurs références à ST(n), où n est la plupart du temps 0. "ST" est en fait une macro qui désigne le n-ième argument sur la pile. ST(0) est donc le premier argument passé à la XSUB, ST(1) le second, et ainsi de suite.

Quand vous faites la liste des arguments de la XSUB dans le fichier .xs, vous dites à xsubpp à quel argument correspond chaque élément de la pile (le premier dans la liste est le premier argument, et ainsi de suite). Vous vous exposez au pire si vous ne les énumérez pas dans l'ordre où la fonction les attend.

### 57.1.17 ETENDRE VOTRE EXTENSION

Parfois, vous souhaitez peut-être fournir des méthodes ou des sous-routines supplémentaires pour vous permettre de rendre l'interface entre Perl et votre extension plus simple ou plus facile à comprendre. Ces routines devraient être implémentées dans le fichier .pm. C'est l'emplacement de leur définition à l'intérieur du fichier .pm qui détermine si elles sont chargées automatiquement en même temps que l'extension elle-même, ou seulement quand on les appelle.

### 57.1.18 DOCUMENTATION DE VOTRE EXTENSION

Aucune excuse ne peut vous dispenser de rédiger une documentation pour de votre extension. Sa place est dans le fichier .pm. Ce fichier sera passé à pod2man, et la documentation intégrée sera convertie au format manpage, puis placée dans le répertoire blib. Elle sera enfin copiée dans le répertoire man de Perl lors de l'installation de l'extension.

Vous pouvez entremêler la documentation et le code Perl dans votre fichier .pm. En fait, vous y serez forcé si vous comptez utiliser l'autochargement des méthodes, comme l'explique un commentaire dans le fichier .pm.

Pour plus d'informations sur le format pod, consultez *perlpod*.

### 57.1.19 INSTALLATION DE VOTRE EXTENSION

Une fois que votre extension est complète et qu'elle a passé tous ses tests avec succès, l'installation est plutôt simple : vous avez juste à lancer "make install". Vous devrez avoir des droits en écriture dans les répertoires où Perl est installé, ou bien vous devrez demander à votre administrateur système de lancer le make à votre place.

### 57.1.20 VOIR AUSSI

Pour plus d'informations, consultez *perlguts*, *perlx*, *perlmod*, et *perlpod*.

## 57.2 AUTEUR

Jeff Okamoto <okamoto@corp.hp.com>

Revu et assisté par Dean Roehrich, Ilya Zakharevich, Andreas Koenig, et Tim Bunce.

### 57.2.1 Date de dernière modification

1996/7/10 pour la version originale

## 57.3 TRADUCTION

### 57.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 57.3.2 Traducteur

Thierry Bézecourt <thbzcr@mail.com>.

### 57.3.3 Relecture

Personne pour l'instant.

# Chapitre 58

## perlx

Manuel de référence du langage XS

### 58.1 DESCRIPTION

#### 58.1.1 Introduction

XS est un langage utilisé pour réaliser une interface d'extension entre Perl et une librairie C qu'on souhaite utiliser depuis Perl. L'interface XS est combinée avec la librairie pour produire une nouvelle librairie susceptible d'être liée avec Perl. Une **XSUB** est une fonction écrite dans le langage XS ; c'est le composant central de l'interface de l'application Perl.

Le compilateur XS s'appelle **xsubpp**. Ce compilateur insère les constructions nécessaires pour permettre à une XSUB, qui n'est autre qu'une fonction C déguisée, de manipuler des valeurs Perl, et crée la colle nécessaire pour permettre à Perl d'accéder à la XSUB. Le compilateur utilise des **typemaps** pour faire la correspondance entre les variables et les paramètres de fonction C d'une part, et les valeurs Perl de l'autre. Le typemap par défaut gère de nombreux types C parmi les plus courants. Un typemap supplémentaire doit être créé pour traiter les structures et les types spécifiques à la librairie que l'on veut lier dans Perl.

Consultez *perlxstut* pour un guide d'apprentissage du processus de création d'extensions dans son ensemble.

Note : pour beaucoup d'extensions, le système SWIG de Dave Beazley fournit un mécanisme nettement plus pratique pour réaliser le code de liaison XS. Voyez <http://www.cs.utah.edu/~beazley/SWIG> pour plus d'information.

#### 58.1.2 Mise en route

La plupart des exemples qui suivent portent sur la création d'une interface entre Perl et les fonctions de la librairie de connexion ONC+ RPC. La fonction `rpcb_gettime()` sera utilisée pour illustrer de nombreuses caractéristiques du langage XS. Cette fonction accepte deux paramètres ; le premier est une donnée en entrée et le second une donnée en sortie. La fonction renvoie aussi une valeur d'état.

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

Depuis C, cette fonction est appelée avec les instructions suivantes.

```
#include <rpc/rpc.h>
bool_t status;
time_t timep;
status = rpcb_gettime("localhost", &timep);
```

Si on réalise une XSUB offrant une traduction directe entre cette fonction et Perl, cette XSUB sera utilisée depuis Perl avec le code suivant. Les variables `$status` et `$timep` contiendront la sortie de cette fonction.

```
use RPC;
$status = rpcb_gettime("localhost", $timep);
```

Le fichier XS suivant présente une sous-routine XS, ou XSUB, proposant un exemple d'interface avec la fonction `rpcb_gettime()`. Cette XSUB représente une traduction directe entre C et Perl et préserve donc l'interface même depuis Perl. Cette XSUB sera appelée depuis Perl de la même manière que ci-dessus. Notez que les trois premières instructions `#include`, pour `EXTERN.h`, `perl.h` et `XSUB.h`, seront tout le temps présentes au début d'un fichier XS. On élargira plus tard cette approche, et on en présentera d'autres.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <rpc/rpc.h>

MODULE = RPC PACKAGE = RPC

bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
 OUTPUT:
 timep
```

Toutes les extensions à Perl, y compris celles qui contiennent des XSUB, doivent être accompagnées d'un module Perl d'amorçage (angl. *bootstrap*) qui assure l'intégration de l'extension dans Perl. Ce module exporte les fonctions et les variables de l'extension vers le programme Perl et entraîne la liaison entre les XSUB de l'extension et Perl. Le module qui suit sera utilisé dans la plupart des exemples de ce document ; il doit être utilisé depuis Perl avec la commande `use` comme on l'a vu précédemment. On approfondira l'étude des modules Perl plus loin dans ce document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime);

bootstrap RPC;
1;
```

Tout au long de ce document, diverses interfaces à la XSUB `rpcb_gettime()` seront explorées. L'ordre ou le nombre des paramètres pris par les XSUB variera. Dans chaque cas, la XSUB est une abstraction entre Perl et la vraie fonction C `rpcb_gettime()`, et la XSUB doit toujours s'assurer que la vraie fonction `rpcb_gettime()` est appelée avec les bons paramètres. Cette abstraction permettra au programmeur de réaliser une interface avec la fonction C plus proche de Perl.

### 58.1.3 Anatomie d'une XSUB

La XSUB qui suit permet à un programme Perl d'accéder à une fonction de librairie C dont le nom est `sin()`. La XSUB fait comme la fonction C, qui prend un seul paramètre et renvoie une valeur unique.

```
double
sin(x)
 double x
```

Lorsqu'on utilise des pointeurs C, l'opérateur d'indirection `*` devrait être considéré comme une partie du type et l'opérateur d'adressage `&` comme une partie de la variable, comme on l'a fait dans la fonction `rpcb_gettime()` ci-dessus. Consultez la section sur les typemaps pour avoir plus de détails sur l'utilisation des qualificatifs et des opérateurs unaires dans les types C.

Le nom de la fonction et le type de retour doivent être placés sur des lignes séparées.

INCORRECT

CORRECT

double sin(x)	double
double x	sin(x)
	double x

Le corps de la fonction peut être mis en retrait ou ajusté à gauche. L'exemple suivant montre une fonction dont le corps est ajusté à gauche. On utilisera un retrait dans la plupart des exemples de ce document.

CORRECT

```
double
sin(x)
double
```

#### 58.1.4 La pile des arguments

La pile des arguments est utilisée pour stocker les valeurs qui sont envoyées à la XSUB en tant que paramètres, ainsi que la valeur de retour de la XSUB. En fait, toutes les fonctions Perl mettent leurs valeurs sur cette pile en même temps, chacune étant limitée à son propre intervalle de positions sur la pile. Dans ce document, la première position dans la pile appartenant à la fonction active sera désignée comme la position 0 pour cette fonction.

Les XSUB se réfèrent à leurs arguments de la pile avec la macro **ST(x)**, où *x* désigne une position dans l'intervalle appartenant à la XSUB sur la pile. La position 0 pour cette fonction est connue de la XSUB comme ST(0). Les paramètres en entrée et les valeurs de retour de la XSUB commencent toujours à ST(0). Dans de nombreux cas simples, le compilateur **xsubpp** générera le code nécessaire à la manipulation de la pile des arguments en insérant des morceaux de code trouvés dans les typemaps. Dans les cas plus complexes, le programmeur devra fournir le code.

#### 58.1.5 La variable RETVAL

La variable **RETVAL** est une variable magique qui est toujours du type retourné par la fonction de la librairie C. Le compilateur **xsubpp** fournit cette variable dans chaque XSUB et l'utilise par défaut pour y mettre la valeur de retour de la fonction C appelée. Dans les cas simples, la valeur de **RETVAL** sera mise dans ST(0) sur la pile d'arguments, où Perl le recevra comme valeur de retour de la XSUB.

Si la XSUB a un type de retour égal à **void**, le compilateur ne fournira pas de variable **RETVAL** pour cette fonction. Lorsqu'on utilise la directive **PPCODE:**, la variable **RETVAL** n'est plus nécessaire, sauf si on l'utilise explicitement.

Si la directive **PPCODE:** n'est pas utilisée, la valeur de retour **void** devrait être utilisée uniquement pour des sous-routines qui ne renvoient pas de valeur, *même si* la directive **CODE:** est utilisée pour positionner ST(0) explicitement.

Dans des versions plus anciennes de ce document, on conseillait d'utiliser la valeur de retour **void** dans de tels cas. On a découvert que cela pouvait mener à des segfaults lorsque la XSUB était *vraiment* **void**. Cette pratique est maintenant désapprouvée, et risque de ne plus être supportée dans une version future. Utilisez la valeur de retour **SV \*** dans ces cas-là (**xsubpp** contient actuellement du code heuristique qui tente de faire la différence entre les fonctions "vraiment-void" et celles qui sont "déclarées-void-suivant-l'ancienne-pratique"; votre code est donc à la merci de l'heuristique si vous n'utilisez pas **SV \*** comme valeur de retour).

#### 58.1.6 Le mot-clé MODULE

Le mot-clé **MODULE** est utilisé pour marquer le début du code XS et spécifier un paquetage pour les fonctions que l'on est en train de définir. Tout le texte qui précède le premier mot-clé **MODULE** est considéré comme du code C et sera transféré dans la sortie du compilateur sans modification. Tout module XS aura une fonction d'initialisation utilisée pour brancher la XSUB dans Perl. Le nom du paquetage de cette fonction d'initialisation correspondra à la valeur de la dernière instruction **MODULE** dans les fichiers source XS. La valeur de **MODULE** devrait toujours rester constante à l'intérieur d'un même fichier XS, même si cela n'est pas obligatoire.

L'exemple suivant démarre le code XS et place toutes les fonctions dans un paquetage nommé **RPC**.

```
MODULE = RPC
```



### 58.1.7 Le mot-clé PACKAGE

Lorsque les fonctions, à l'intérieur d'un fichier source XS, doivent être réparties dans des paquetages, il faut utiliser le mot-clé PACKAGE. Ce mot-clé est utilisé avec le mot-clé MODULE et doit être spécifié immédiatement après lui.

```
MODULE = RPC PACKAGE = RPC

[code XS dans le paquetage RPC]

MODULE = RPC PACKAGE = RPCB

[code XS dans le paquetage RPCB]

MODULE = RPC PACKAGE = RPC

[code XS dans le paquetage RPC]
```

Bien que ce mot-clé soit optionnel et qu'il fournisse des informations redondantes dans certains cas, il devrait toujours être utilisé. Il permet de garantir que la XSUB apparaîtra dans le paquetage désiré.

### 58.1.8 Le mot-clé PREFIX

Le mot-clé PREFIX désigne des préfixes à supprimer dans les noms de fonction Perl. Si la fonction C est `rpcb_gettime()` et que la valeur de PREFIX est `rpcb_`, Perl devra voir cette fonction comme `_gettime()`.

Ce mot-clé suit le mot-clé PACKAGE lorsqu'il est utilisé. Si PACKAGE n'est pas utilisé, alors PREFIX suit le mot-clé MODULE.

```
MODULE = RPC PREFIX = rpcb_

MODULE = RPC PACKAGE = RPCB PREFIX = rpcb_
```

### 58.1.9 Le mot-clé OUTPUT

Le mot-clé OUTPUT: indique qu'il faut mettre à jour certains paramètres de fonction (en rendant les nouvelles valeurs accessibles à Perl) lorsque la XSUB se termine, ou que certaines valeurs doivent être renvoyées à la fonction Perl. Pour des fonctions simples, comme la fonction `sin()` ci-dessus, la variable RETVAL est désignée automatiquement comme une valeur en sortie. Dans des fonctions plus complexes, le compilateur **xsubpp** aura besoin d'aide pour déterminer quelles variables sont des variables en sortie.

Ce mot-clé sera utilisé normalement en complément du mot-clé CODE:. La variable RETVAL n'est pas reconnue comme une variable en sortie lorsque le mot-clé CODE: est présent. Le mot-clé OUTPUT: est utilisé dans cette situation pour dire au compilateur que RETVAL est réellement une variable en sortie.

Le mot-clé OUTPUT: peut aussi être utilisé pour indiquer que des paramètres de fonction sont des variables en sortie. Cela peut être nécessaire lorsqu'un paramètre a été modifié à l'intérieur d'une fonction et que le programmeur veut que cette modification soit visible depuis Perl.

```
bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
OUTPUT:
 timep
```

Le mot-clé OUTPUT: permet aussi à un paramètre en sortie d'être relié à un morceau de code correspondant plutôt qu'à un typemap.

```

bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
OUTPUT:
 timep sv_setnv(ST(1), (double)timep);

```

**xsubpp** rajoute automatiquement un `SvSETMAGIC()` pour tous les paramètres de la section OUTPUT de la XSUB, sauf pour RETVAL. C'est le comportement désiré la plupart du temps, car il se charge d'invoquer convenablement l'effet 'set' (angl. "set magic") sur les données en sortie (ce qui est nécessaire pour les éléments de tableau simple ou de tableau associatif passés en paramètre, qui doivent être créés s'ils n'existent pas). Si, pour une raison donnée, ce comportement n'est pas désiré, la section OUTPUT peut contenir une ligne `SETMAGIC: DISABLE` qui le désactive pour le restant des paramètres de la section OUTPUT:. De même, `SETMAGIC: ENABLE` peut être utilisé pour le réactiver pour le restant de la section OUTPUT. Voyez *perlguts* pour plus de détails sur l'effet 'set'.

### 58.1.10 Le mot-clé CODE

Ce mot-clé est utilisé dans des XSUB plus complexes qui nécessitent un traitement spécial pour la fonction C. La variable RETVAL est disponible, mais elle ne sera pas retournée, sauf si elle est spécifiée sous le mot-clé OUTPUT:.

La XSUB qui suit correspond à une fonction C qui requiert un traitement spécial de ses paramètres. L'utilisation depuis Perl est donnée en premier.

```
$status = rpcb_gettime("localhost", $timep);
```

Voici la XSUB.

```

bool_t
rpcb_gettime(host,timep)
 char *host
 time_t timep
CODE:
 RETVAL = rpcb_gettime(host, &timep);
OUTPUT:
 timep
 RETVAL

```

### 58.1.11 Le mot-clé INIT

Le mot-clé INIT: permet d'insérer du code d'initialisation dans la XSUB avant que le compilateur ne génère l'appel à la fonction C. Contrairement au mot-clé CODE: ci-dessus, celui-ci n'affecte pas la manière dont le compilateur traite RETVAL.

```

bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
INIT:
 printf("# l'hote est %s\n", host);
OUTPUT:
 timep

```

### 58.1.12 Le mot-clé NO\_INIT

Le mot-clé NO-INIT sert à indiquer qu'on utilise un paramètre de la fonction uniquement comme valeur en sortie. Le compilateur **xsubpp** génère normalement du code pour lire les valeurs de tous les paramètres de la fonction sur la pile des arguments, et pour les assigner à des variables C lors de l'entrée dans la fonction. NO\_INIT dit au compilateur que certains paramètres seront utilisés en sortie plutôt qu'en entrée, et qu'ils seront traités avant la fin de la fonction.

L'exemple suivant présente une variante de la fonction `rpcb_gettime()`. Cette fonction utilise la variable `timep` uniquement comme variable en sortie, et ne se préoccupe pas de son contenu initial.

```

bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep = NO_INIT
OUTPUT:
 timep

```

### 58.1.13 Initialisation des arguments de fonction

Normalement, les paramètres de fonction sont initialisés avec les valeurs qu'ils ont dans la pile des arguments. Les typemaps contiennent les portions de code utilisées pour transférer les valeurs Perl dans des paramètres C. Le programmeur, toutefois, est autorisé à surcharger les typemaps et à fournir un code d'initialisation différent (ou supplémentaire).

Le code qui suit montre comment fournir du code d'initialisation pour les paramètres de fonction. Le code d'initialisation est évalué entre des guillemets par le compilateur (en utilisant `eval()`), avant d'être inséré dans le code généré, de sorte que toute expression devant être interprétée littéralement (essentiellement `$`, `@` ou `\`) doit être protégée par des `\`. Les variables `$var`, `$arg` et `$type` peuvent être utilisées comme dans les typemaps.

```

bool_t
rpcb_gettime(host,timep)
 char *host = (char *)SvPV($arg,PL_na);
 time_t &timep = 0;
OUTPUT:
 timep

```

Ce procédé ne devrait pas être utilisé pour fournir des valeurs par défaut aux paramètres. Le cas normal d'utilisation est celui où un paramètre de fonction doit être traité par une autre fonction de la librairie avant d'être utilisé. Les paramètres par défaut font l'objet de la prochaine section.

Si l'initialisation commence par `=`, elle sera insérée sur la même ligne que la déclaration de la variable. Si elle commence par `;` ou `+`, elle sera insérée une fois que toutes les variables en entrée auront été déclarées. Les cas `=` et `;` remplacent l'initialisation fournie normalement par le typemap. Dans le cas `+`, l'initialisation du typemap précédera le code d'initialisation qui suit le `+`. Une variable globale, `%v`, est disponible pour le cas vraiment rare où une initialisation a besoin d'une information venant d'une autre initialisation.

```

bool_t
rpcb_gettime(host,timep)
 time_t &timep ; /*\${time}=@[${v}{time}=$arg]*/
 char *host + SvOK($v{time}) ? SvPV($arg,PL_na) : NULL;
OUTPUT:
 timep

```

### 58.1.14 Valeurs de paramètres par défaut

On peut donner des valeurs par défaut à des paramètres de fonction en rajoutant une instruction d'affectation dans la liste des paramètres. La valeur par défaut peut être un nombre ou une chaîne de caractères. Les valeurs par défaut doivent toujours être utilisées uniquement pour les paramètres les plus à droite.

Afin de permettre à la XSUB de `rpcb_gettime()` d'avoir un hôte par défaut, les paramètres de la XSUB pourraient être réordonnés. La XSUB appellera alors la vraie fonction `rpcb_gettime()` avec les paramètres placés dans le bon ordre. Perl appellera cette XSUB avec l'une des instructions suivantes.

```
$status = rpcb_gettime($timep, $host);
```

```
$status = rpcb_gettime($timep);
```

La XSUB ressemblera au code qui suit. Un bloc `CODE:` est utilisé pour appeler la vraie fonction `rpcb_gettime()` avec les paramètres remis dans l'ordre correct pour cette fonction.

```

bool_t
rpcb_gettime(timep,host="localhost")
 char *host
 time_t timep = NO_INIT
 CODE:
 RETVAL = rpcb_gettime(host, &timep);
 OUTPUT:
 timep
 RETVAL

```

### 58.1.15 Le mot-clé PREINIT

Le mot-clé PREINIT: permet de déclarer des variables supplémentaires avant que les typemaps soient utilisés. Si une variable est déclarée dans un bloc CODE:, elle suivra tout code généré par le typemap. Cela peut provoquer des fautes de syntaxe en C. Pour forcer la déclaration de la variable avant le code du typemap, placez-la dans un bloc PREINIT:. Le mot-clé PREINIT: peut être utilisé une ou plusieurs fois à l'intérieur d'une XSUB.

Les exemples qui suivent sont équivalents, mais, si le code utilise des typemaps complexes, le premier exemple sera plus sûr.

```

bool_t
rpcb_gettime(timep)
 time_t timep = NO_INIT
 PREINIT:
 char *host = "localhost";
 CODE:
 RETVAL = rpcb_gettime(host, &timep);
 OUTPUT:
 timep
 RETVAL

```

Un exemple correct, mais avec des risques d'erreur.

```

bool_t
rpcb_gettime(timep)
 time_t timep = NO_INIT
 CODE:
 char *host = "localhost";
 RETVAL = rpcb_gettime(host, &timep);
 OUTPUT:
 timep
 RETVAL

```

### 58.1.16 Le mot-clé SCOPE

Le mot-clé SCOPE: permet d'activer un niveau spécial de visibilité (angl. scoping) pour une XSUB donnée. Dans ce cas-là, la XSUB invoquera ENTER et LEAVE automatiquement (Note du Traducteur : les macros ENTER et LEAVE sont décrites dans *perlcall*).

Afin de supporter des correspondances de type complexes, ce niveau spécial sera activé automatiquement pour une XSUB si elle utilise une entrée de typemap contenant le commentaire */\*scope\*/*.

Pour activer ce niveau spécial :

```
SCOPE: ENABLE
```

Pour le désactiver :

```
SCOPE: DISABLE
```

### 58.1.17 Le mot-clé INPUT

Habituellement, les paramètres de XSUB sont évalués juste après l'entrée dans la XSUB. Le mot-clé INPUT: peut être utilisé pour forcer ces paramètres à être évalués un peu plus tard. On peut utiliser le mot-clé INPUT: plusieurs fois dans une XSUB, et ceci pour une ou plusieurs variables en entrée. Ce mot-clé accompagne le mot-clé PREINIT:.

L'exemple suivant montre comment l'évaluation du paramètre en entrée `timep` peut être retardée, après un PREINIT:

```
bool_t
rpcb_gettime(host,timep)
 char *host
 PREINIT:
 time_t tt;
 INPUT:
 time_t timep
 CODE:
 RETVAL = rpcb_gettime(host, &tt);
 timep = tt;
 OUTPUT:
 timep
 RETVAL
```

Dans cet autre exemple, chacun des paramètres en entrée voit son évaluation retardée.

```
bool_t
rpcb_gettime(host,timep)
 PREINIT:
 time_t tt;
 INPUT:
 char *host
 PREINIT:
 char *h;
 INPUT:
 time_t timep
 CODE:
 h = host;
 RETVAL = rpcb_gettime(h, &tt);
 timep = tt;
 OUTPUT:
 timep
 RETVAL
```

### 58.1.18 Listes de paramètres de longueur variable

Les XSUB peuvent recevoir des listes de paramètres de longueur variable en spécifiant une ellipse `...` dans la liste des paramètres. Cette utilisation de l'ellipse est similaire à celle que l'on trouve en C ANSI. Le programmeur peut déterminer le nombre d'arguments passés à la XSUB en examinant la variable `items` que le compilateur `xsubpp` fournit pour chaque XSUB. Grâce à ce mécanisme, on peut réaliser une XSUB qui accepte une liste de paramètres de longueur indéterminée.

Le paramètre `host` de la XSUB `rpcb_gettime()` peut être optionnel, de manière à ce qu'on puisse utiliser l'ellipse pour indiquer que la XSUB prendra un nombre variable de paramètres. Perl devrait pouvoir appeler cette XSUB avec chacune des instructions suivantes.

```
$status = rpcb_gettime($timep, $host);
```

```
$status = rpcb_gettime($timep);
```

Voici le code XS, avec l'ellipse :

```

bool_t
rpcb_gettime(timep, ...)
 time_t timep = NO_INIT
 PREINIT:
 char *host = "localhost";
 CODE:
 if(items > 1)
 host = (char *)SvPV(ST(1), PL_na);
 RETVAL = rpcb_gettime(host, &timep);
 OUTPUT:
 timep
 RETVAL

```

### 58.1.19 Le mot-clé C\_ARGS

Le mot-clé C\_ARGS permet de réaliser des XSUB que l'on n'appelle pas de la même manière depuis Perl que depuis C, sans qu'il soit nécessaire d'écrire une section CODE: ou PPCODE:. Le contenu du paragraphe C\_ARGS est passé comme argument à la fonction C, sans aucun changement.

Supposons par exemple que la fonction C soit déclarée ainsi :

```
symbolic nth_derivative(int n, symbolic function, int flags);
```

et que la valeur par défaut de *flags* soit contenue dans la variable C *default\_flags*. Supposons que vous souhaitiez réaliser une interface que l'on appellera de la manière suivante :

```
$second_deriv = $function->nth_derivative(2);
```

Pour cela, déclarez la XSUB ainsi :

```

symbolic
nth_derivative(function, n)
 symbolic function
 int n
C_ARGS:
 n, function, default_flags

```

### 58.1.20 Le mot-clé PPCODE

Le mot-clé PPCODE: est une variante du mot-clé CODE: qui est utilisée pour indiquer au compilateur **xsubpp** que le programmeur fournit le code contrôlant la pile des arguments pour les valeurs de retour des XSUB. On souhaite parfois que la XSUB renvoie une liste de valeurs et non une valeur unique. Dans ce cas-là, il faut utiliser PPCODE: et rajouter de manière explicite la liste des valeurs sur la pile. Les mots-clés PPCODE: et CODE: ne sont pas utilisés simultanément dans la même XSUB.

La XSUB suivante appelle la fonction C `rpcb_gettime()` et renvoie à Perl ses deux valeurs de sortie, `timep` et `status`, comme une seule liste.

```

void
rpcb_gettime(host)
 char *host
 PREINIT:
 time_t timep;
 bool_t status;
 PPCODE:
 status = rpcb_gettime(host, &timep);
 EXTEND(SP, 2);
 PUSHs(sv_2mortal(newSViv(status)));
 PUSHs(sv_2mortal(newSViv(timep)));

```

Remarquez que le programmeur doit fournir le code C assurant l'appel de la vraie fonction `rpcb_gettime()`, ainsi que le placement correct des valeurs de retour sur la pile des arguments.

Le type de retour `void` pour cette fonction indique au compilateur **xsubpp** que la variable `RETVAL` n'est pas nécessaire, qu'elle n'est pas utilisée, et qu'elle ne devrait pas être créée. Dans la plupart des cas, il faut utiliser le type de retour `void` avec l'instruction `PPCODE`:

On utilise la macro `EXTEND()` afin de dégager de la place sur la pile des arguments pour les 2 valeurs de retour. L'instruction `PPCODE`: fait en sorte que le compilateur **xsubpp** crée un pointeur vers la pile dans la variable `SP`; c'est ce pointeur qui est utilisé dans la macro `EXTEND()`. Les valeurs sont ensuite rajoutées sur la pile avec les macros `PUSHs()`.

A présent, la fonction `rpcb_gettime()` peut être utilisée depuis Perl avec l'instruction suivante.

```
($status, $timep) = rpcb_gettime("localhost");
```

Lorsque vous travaillez sur les paramètres en sortie avec une section `PPCODE`:, assurez-vous de traiter l'effet 'set' convenablement. Consultez *perlguts* pour plus de détails sur cet effet 'set'.

### 58.1.21 Renvoyer undef et des listes vides

Le programmeur voudra parfois renvoyer simplement `undef` ou une liste vide si la fonction échoue, plutôt qu'une valeur d'état à part. La fonction `rpcb_gettime()` présente justement cette situation. Nous aimerions que la fonction retourne l'heure si elle réussit, ou `undef` si elle échoue. Dans le code Perl suivant, la valeur de `$timep` sera soit `undef`, soit une heure valide;

```
$timep = rpcb_gettime("localhost");
```

La XSUB suivante n'utilise le type de retour `SV *` qu'à titre d'aide-mémoire, et il utilise un bloc `CODE`: pour indiquer au compilateur que le programmeur a fourni tout le code nécessaire. L'appel de `sv_newmortal()` initialise la valeur de retour à `undef`, ce qui en fait la valeur de retour par défaut.

```
SV *
rpcb_gettime(host)
 char * host
 PREINIT:
 time_t timep;
 bool_t x;
 CODE:
 ST(0) = sv_newmortal();
 if(rpcb_gettime(host, &timep))
 sv_setnv(ST(0), (double)timep);
```

Cet autre exemple montre comment on peut mettre un `undef` explicite dans la valeur de retour, au cas où le besoin s'en ferait ressentir.

```
SV *
rpcb_gettime(host)
 char * host
 PREINIT:
 time_t timep;
 bool_t x;
 CODE:
 ST(0) = sv_newmortal();
 if(rpcb_gettime(host, &timep)){
 sv_setnv(ST(0), (double)timep);
 }
 else{
 ST(0) = &PL_sv_undef;
 }
```

Pour renvoyer une liste vide, il faut utiliser un bloc `PPCODE`: et ne pas rajouter de valeur de retour sur la pile.

```

void
rpcb_gettime(host)
 char *host
 PREINIT:
 time_t timep;
 PPCODE:
 if(rpcb_gettime(host, &timep))
 PUSHs(sv_2mortal(newSViv(timep)));
 else{
 /* Rien n'est remis sur la pile, donc une */
 /* liste vide est renvoyee implicitement */
 }

```

Certaines personnes préfèrent rajouter un `return` explicite dans la XSUB ci-dessus au lieu de laisser l'exécution se poursuivre jusqu'au bout. Dans ce cas-là, il convient plutôt d'utiliser `XSRETURN_EMPTY`, ce qui garantira que la pile XSUB est ajustée correctement. D'autres macros sont décrites dans *perlguts*.

### 58.1.22 Le mot-clé REQUIRE

Le mot-clé `REQUIRE`: sert à indiquer le numéro de version minimal du compilateur `xsubpp` requis pour compiler le module XS. Un module XS contenant l'instruction suivante ne pourra être compilé qu'avec une version de `xsubpp` égale ou supérieure à 1.922 :

```
REQUIRE: 1.922
```

### 58.1.23 Le mot-clé CLEANUP

Ce mot-clé peut être utilisé quand une XSUB doit exécuter des procédures de nettoyage spéciales avant de se terminer. Quand le mot-clé `CLEANUP`: est utilisé, il doit suivre tout bloc `CODE:`, `PPCODE:` ou `OUTPUT:` présent dans la XSUB. Les instructions spécifiées dans le bloc de nettoyage seront les dernières instructions de la XSUB.

### 58.1.24 Le mot-clé BOOT

Le mot-clé `BOOT`: permet de rajouter du code dans la fonction d'amorçage de l'extension. La fonction d'amorçage est générée par le compilateur `xsubpp` et contient normalement les instructions nécessaires pour enregistrer toute XSUB auprès de Perl. Avec le mot-clé `BOOT:`, le programmeur peut dire au compilateur de rajouter des instructions supplémentaires dans la fonction d'amorçage.

Ce mot-clé peut être utilisé n'importe quand après le premier mot-clé `MODULE`, et doit être tout seul sur une ligne. La première ligne blanche après le mot-clé terminera le bloc de code.

```

BOOT:
Le message suivant sera affiche lors de l'execution de la
fonction d'amorçage.
printf("bonjour !\n");

```

### 58.1.25 Le mot-clé VERSIONCHECK

Le mot-clé `VERSIONCHECK`: correspond aux options `-versioncheck` et `-noverioncheck` de `xsubpp`. Ce mot-clé prime sur les options de la ligne de commande. La vérification de version est activée par défaut. Lorsqu'elle est activée, le module XS essaie de vérifier que son numéro de version est compatible avec celui du module PM.

Pour activer la vérification de version :

```
VERSIONCHECK: ENABLE
```

Pour la désactiver :

```
VERSIONCHECK: DISABLE
```



### 58.1.26 Le mot-clé PROTOTYPES

Le mot-clé PROTOTYPES: correspond aux options `-prototypes` et `-noprototypes` de **xsubpp**. Ce mot-clé prime sur les options de la ligne de commande. Les prototypes sont activés par défaut. Lorsqu'ils sont activés, les XSUB reçoivent des prototypes Perl. Ce mot-clé peut être utilisé plusieurs fois dans un module XS pour activer et désactiver les prototypes dans différentes parties du module.

Pour activer les prototypes :

```
PROTOTYPES: ENABLE
```

Pour les désactiver :

```
PROTOTYPES: DISABLE
```

### 58.1.27 Le mot-clé PROTOTYPE

Ce mot-clé est proche du mot-clé PROTOTYPES: ci-dessus, mais peut être utilisé pour forcer **xsubpp** à utiliser un prototype donné pour la XSUB. Ce mot-clé prime sur toute autre option ou mot-clé relatif au prototypage, mais n'affecte que la XSUB courante. Consultez *Prototypes* in *perlsb* pour plus d'informations sur les prototypes Perl.

```
bool_t
rpcb_gettime(timep, ...)
 time_t timep = NO_INIT
 PROTOTYPE: $;$
 PREINIT:
 char *host = "localhost";
 CODE:
 if(items > 1)
 host = (char *)SvPV(ST(1), PL_na);
 RETVAL = rpcb_gettime(host, &timep);
 OUTPUT:
 timep
 RETVAL
```

### 58.1.28 Le mot-clé ALIAS

Le mot-clé ALIAS: permet à une XSUB de recevoir plusieurs noms en Perl, et de déterminer lequel de ces noms a été utilisé pour l'invoquer. Les noms en Perl peuvent être complets avec le nom du paquetage. Chaque alias reçoit un numéro. Le compilateur crée une variable nommée `ix` qui contient le numéro de l'alias utilisé. Lorsque le nom utilisé pour invoquer la XSUB est celui avec lequel il a été déclaré, `ix` est égal à 0.

L'exemple suivant crée des alias `FOO::_gettime()` et `BAR::gett()` pour cette fonction.

```
bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
 ALIAS:
 FOO::_gettime = 1
 BAR::gett = 2
 INIT:
 printf("# ix = %d\n", ix);
 OUTPUT:
 timep
```

### 58.1.29 Le mot-clé INTERFACE

Ce mot-clé déclare que la XSUB courante sert à garder en réserve la signature de fonction indiquée. S'il est suivi par du texte, ce texte est considéré comme une liste de fonctions qui ont cette signature, et qui doivent être attachées à des XSUB. Par exemple, si vous avez 4 fonctions `multiply()`, `divide()`, `add()` et `subtract()`, toutes avec la signature

```
symbolic f(symbolic, symbolic);
```

vous pouvez les implémenter toutes à la fois avec la XSUB

```
symbolic
interface_s_ss(arg1, arg2)
 symbolic arg1
 symbolic arg2
INTERFACE:
 multiply divide
 add subtract
```

L'avantage de cette approche par rapport au mot-clé ALIAS: est que l'on peut attacher une fonction supplémentaire `remainder()` lors de l'exécution en utilisant

```
CV *mycv = newXSproto("Symbolic::remainder",
 XS_Symbolic_interface_s_ss, __FILE__, "$$");
XSINTERFACE_FUNC_SET(mycv, remainder);
```

(on suppose dans cet exemple qu'il n'y a pas de section `INTERFACE_MACRO:`, car il faut alors utiliser autre chose que `XSINTERFACE_FUNC_SET`)

### 58.1.30 Le mot-clé INTERFACE\_MACRO

Ce mot-clé permet de définir une `INTERFACE` qui procède d'une manière différente pour récupérer dans la XSUB un pointeur vers la fonction C. Le texte qui suit ce mot-clé doit indiquer des noms de macros à utiliser pour récupérer le pointeur de fonction ou pour l'initialiser dans la XSUB. La macro de récupération reçoit le type de retour, `CV*`, et `XSANY.any_dptr` pour ce `CV*`. La macro d'initialisation reçoit `cv` et le pointeur de fonction.

Les valeurs par défaut de ces macros sont `XSINTERFACE_FUNC` et `XSINTERFACE_FUNC_SET`. Le mot-clé `INTERFACE` avec une liste vide de fonctions peut être omis si `INTERFACE_MACRO` est utilisé.

Supposons que, dans l'exemple qui précède, des pointeurs de fonction pour `multiply()`, `divide()`, `add()`, et `subtract()` soient conservés dans un tableau C global `fp[]`, les positions de ces pointeurs par rapport au début du tableau étant contenues dans des variables nommées `multiply_off`, `divide_off`, `add_off`, `subtract_off`. On peut alors utiliser

```
#define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
 ((XSINTERFACE_CVT(ret,))fp[CvXSUBANY(cv).any_i32])
#define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
 CvXSUBANY(cv).any_i32 = CAT2(f, _off)
```

dans la section C, et

```
symbolic
interface_s_ss(arg1, arg2)
 symbolic arg1
 symbolic arg2
INTERFACE_MACRO:
 XSINTERFACE_FUNC_BYOFFSET
 XSINTERFACE_FUNC_BYOFFSET_set
INTERFACE:
 multiply divide
 add subtract
```

dans la section XSUB.

### 58.1.31 Le mot-clé INCLUDE

Ce mot-clé peut être utilisé pour insérer d'autres fichiers dans le module XS. Les autres fichiers peuvent contenir du code XS. INCLUDE: peut aussi être utilisé pour exécuter une commande générant du code XS à insérer dans le module.

Le fichier *Rpcb1.xsh* contient notre fonction `rpcb_gettime()` :

```
bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
OUTPUT:
 timep
```

Le module XS peut utiliser INCLUDE: pour insérer ce fichier.

```
INCLUDE: Rpcb1.xsh
```

Si les paramètres du mot-clé INCLUDE: sont suivis d'un trait vertical (|), alors le compilateur interprétera les paramètres comme une commande.

```
INCLUDE: cat Rpcb1.xsh |
```

### 58.1.32 Le mot-clé CASE

Le mot-clé CASE permet à une XSUB de se subdiviser en plusieurs parties distinctes, chacune se comportant comme une XSUB virtuelle. CASE: est global à la XSUB : quand on l'utilise, il faut mettre tous les autres mots-clés XS à l'intérieur d'un CASE:. Cela signifie que rien ne peut précéder le premier CASE: dans la XSUB, et que tout code suivant le dernier CASE: y est incorporé.

Un CASE: peut être associé à une condition sur un paramètre de la XSUB en utilisant la variable-alias `ix` (voir Le mot-clé ALIAS: (§??)), ou éventuellement la variable `items` (voir Listes de paramètres de longueur variable (§58.1.18)). Le dernier CASE correspond au mot-clé **default** en C s'il n'est associé à aucune condition. L'exemple suivant montre un CASE dépendant de `ix` dans la fonction `rpcb_gettime()`, laquelle a pour alias `x_gettime()`. Lorsque la fonction est invoquée sous le nom `rpcb_gettime()`, elle reçoit les paramètres habituels (`char *host, time_t *timep`), tandis que, lorsque elle est invoquée en tant que `x_gettime()`, ses paramètres sont inversés, (`time_t *timep, char *host`).

```
long
rpcb_gettime(a,b)
 CASE: ix == 1
 ALIAS:
 x_gettime = 1
 INPUT:
 # 'a' est timep, 'b' est host
 char *b
 time_t a = NO_INIT
 CODE:
 RETVAL = rpcb_gettime(b, &a);
 OUTPUT:
 a
 RETVAL
 CASE:
 # 'a' est host, 'b' est timep
 char *a
 time_t &b = NO_INIT
 OUTPUT:
 b
 RETVAL
```

Cette fonction peut être appelée avec chacune des instructions suivantes. Notez la différence dans l'ordre des arguments.

```
$status = rpcb_gettime($host, $timep);
```

```
$status = x_gettime($timep, $host);
```

### 58.1.33 L'opérateur unaire &

L'opérateur unaire & est utilisé pour dire au compilateur qu'il doit déréférencer l'objet lors de l'appel à la fonction C. On utilise ce procédé lorsqu'on n'a pas mis de bloc CODE: et que l'objet n'est pas de type pointeur (l'objet est un int ou un long, mais pas un int\* ni un long\*).

La XSUB suivante génère du code C incorrect. Le code généré par **xsubpp** appellera `rpcb_gettime()` avec les paramètres (`char *host, time_t timep`), mais le vrai `rpcb_gettime()` attend un paramètre `timep` de type `time_t *` et non `time_t`.

```
bool_t
rpcb_gettime(host,timep)
 char *host
 time_t timep
OUTPUT:
 timep
```

On résout ce problème en utilisant l'opérateur &. Le compilateur **xsubpp**, à présent, traduira la XSUB en code qui appellera `rpcb_gettime()` de manière correcte, avec les paramètres (`char *host, time_t *timep`). Il le fait en conservant le & tel quel, de sorte que l'appel de fonction est `rpcb_gettime(host, &timep)`.

```
bool_t
rpcb_gettime(host,timep)
 char *host
 time_t &timep
OUTPUT:
 timep
```

### 58.1.34 Insérer des commentaires et des directives de pré-processeur C

Des directives de pré-processeur C peuvent prendre place à l'intérieur des blocs BOOT:, PREINIT:, INIT:, CODE:, PP-CODE: et CLEANUP:, de même qu'en dehors des fonctions. Les commentaires sont autorisés partout après le mot-clé MODULE. Le compilateur transmet les directives de pré-processeur sans modification et supprime les lignes commentées.

On peut rajouter des commentaires dans les XSUB en mettant un # sur une ligne comme premier caractère non blanc. Attention, le commentaire ne doit pas ressembler à une directive de pré-processeur C, sinon il sera interprété comme tel. Le moyen le plus simple d'éviter cela consiste à mettre des caractères blancs devant le #.

Si vous utilisez des directives de pré-processeur pour choisir entre deux versions d'une fonction, utilisez

```
#if ... version1
#else /* ... version2 */
#endif
```

et non pas

```
#if ... version1
#endif
#if ... version2
#endif
```

car, dans le second cas, `xsubpp` croira que vous avez défini la fonction deux fois. Par ailleurs, insérez une ligne blanche avant le `#else` et le `#endif` afin qu'ils ne soient pas considérés comme une partie intégrante du corps de la fonction.

### 58.1.35 Utiliser XS avec C++

Si une fonction est définie comme une méthode C++, alors elle considère son premier argument comme un pointeur vers un objet. Le pointeur vers l'objet est stocké dans une variable nommée THIS. L'objet doit avoir été créé en C++ avec la fonction `new()`, et il doit être béni par Perl (comme avec la fonction `bless()` en Perl pur) avec la macro `sv_setref_pv()`. La *bénédictio*n d'un objet par Perl peut être effectuée par le `typemap`. Un exemple de `typemap` est donné à la fin de cette section.

Si la méthode est définie comme statique, elle appellera la fonction C++ en utilisant la syntaxe `classe::methode()`. Si la méthode n'est pas statique, la fonction sera appelée avec la syntaxe `THIS->methode()`.

Les exemples qui suivent utiliseront la classe C++ que voici :

```

class color {
public:
 color();
 ~color();
 int blue();
 void set_blue(int);

private:
 int c_blue;
};

```

Les XSUB pour les méthodes `blue()` et `set_blue()` sont définies avec le nom de la classe, mais le paramètre pour l'objet (THIS, ou "self") est implicite et il n'est pas mentionné dans la liste.

```

int
color::blue()

void
color::set_blue(val)
 int val

```

Les deux fonctions attendent un objet en premier paramètre. Le compilateur `xsubpp` donne à cet objet le nom `THIS`, et l'utilise pour appeler la méthode spécifiée. Ainsi, dans le code C++ généré, les méthodes `blue()` et `set_blue()` seront appelées de la façon suivante :

```

RETVAL = THIS->blue();

THIS->set_blue(val);

```

Si le nom de la fonction est **DESTROY**, alors la fonction C++ `delete` sera appelée avec `THIS` comme paramètre.

```

void
color::DESTROY()

```

Le code C++ appellera `delete`.

```

delete THIS;

```

Si le nom de la fonction est **new**, la fonction C++ `new` sera appelée pour créer un objet C++ dynamique. La XSUB s'attendra à recevoir comme premier argument le nom de la classe, qui sera stocké dans une variable nommée `CLASS`.

```

color *
color::new()

```

Le code C++ appellera `new` :

```

RETVAL = new color();

```

Voici maintenant un exemple de `typemap` qui pourrait être utilisé dans cet exemple C++.

```

TYPEMAP
color * O_OBJECT

OUTPUT
L'objet Perl est beni dans 'CLASS', qui devrait etre un char*
contenant le nom du paquetage servant a le benir.
O_OBJECT
 sv_setref_pvc($arg, CLASS, (void*)$var);

INPUT
O_OBJECT
 if(sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG))
 $var = ($type)SvIV((SV*)SvRV($arg));
 else{
 warn(\ "${Package}::${func_name()} -- $var n'est pas une reference de SV benie\ ");
 XSRETURN_UNDEF;
 }

```

### 58.1.36 Méthode de construction d'interfaces

Lorsque vous concevez une interface entre Perl et une librairie C, dans de nombreux cas une traduction directe de C vers XS est suffisante. L'interface sera souvent très proche de C et parfois non intuitive, surtout lorsque la fonction C modifie l'un de ses paramètres. Si le programmeur souhaite réaliser une interface de style plus proche de Perl, la méthode suivante peut l'aider à repérer les parties les plus importantes de l'interface.

Repérez les fonctions C qui modifient leurs paramètres. Les XSUB de ces fonctions peuvent retourner à Perl des listes, ou bien, en cas d'échec, undef ou une liste vide.

Repérez les valeurs qui sont utilisées uniquement par les fonctions C et les XSUB. Si le code Perl lui-même n'a pas besoin d'accéder au contenu d'une valeur, alors il n'est peut-être pas nécessaire de fournir une traduction de cette valeur de C en Perl.

Repérez les pointeurs dans la liste de paramètres de la fonction C et dans les valeurs de retour. Certains pointeurs peuvent être traités en XS avec l'opérateur unaire & sur le nom de la variable, tandis que d'autres nécessitent l'utilisation de l'opérateur \* sur le nom du type. En général, il est plus facile de travailler avec l'opérateur &.

Repérez les structures utilisées par les fonctions C. On peut souvent utiliser le typemap T\_PTROBJ pour faire en sorte que ces structures puissent être manipulées par Perl comme des objets bénis.

### 58.1.37 Objets Perl et structures C

Lorsqu'on a affaire à des structures C, il faut choisir soit **T\_PTROBJ**, soit **T\_PTRREF** pour le type XS. Ces deux types sont conçus pour manipuler des pointeurs vers des objets complexes. Le type T\_PTRREF peut être utilisé avec un objet Perl non béni, tandis que le type T\_PTROBJ impose que l'objet soit béni. En utilisant T\_PTROBJ, on peut obtenir une sorte de vérification de type, car la XSUB essaiera de vérifier que l'objet Perl possède le type attendu.

Le code XS qui suit montre la fonction `getnetconfig()` utilisée avec `ONC+ TIRPC`. Cette fonction retourne un pointeur vers une structure C, et son prototype C est donné plus bas. Cet exemple montrera comment le pointeur C devient une référence Perl. Perl considérera cette référence comme un pointeur sur l'objet béni, et cherchera à appeler un destructeur pour cet objet. Un destructeur sera fourni dans le code XS pour libérer la mémoire utilisée par `getnetconfig()`. En XS, les destructeurs peuvent être créés en spécifiant une fonction XSUB dont le nom se termine avec le mot **DESTROY**. Les destructeurs XS peuvent être utilisés pour libérer de la mémoire qui aurait été allouée par `malloc()` dans une autre XSUB.

```
struct netconfig *getnetconfig(const char *netid);
```

Un `typedef` peut être créé pour `struct netconfig`. L'objet Perl sera béni dans une classe dont le nom correspondra au type C, avec un suffixe `Ptr`; le nom ne doit pas contenir de blanc s'il doit devenir un nom de paquetage Perl. Le destructeur de l'objet sera mis dans une classe correspondant à la classe de l'objet, et le mot-clé `PREFIX` sera utilisé pour réduire le nom au seul mot `DESTROY`, comme prévu dans Perl.

```
typedef struct netconfig Netconfig;

MODULE = RPC PACKAGE = RPC

Netconfig *
getnetconfig(netid)
 char *netid

MODULE = RPC PACKAGE = NetconfigPtr PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
 Netconfig *netconf
CODE:
 printf("Maintenant dans NetconfigPtr::DESTROY\n");
 free(netconf);
```

Cet exemple requiert l'entrée de typemap suivante. Consultez la section sur les typemaps pour plus d'informations sur l'ajout de nouveaux typemaps dans une extension.

```
TYPEMAP
Netconfig * T_PTROBJ
```

Cet exemple sera utilisé avec l'instruction Perl suivante.

```
use RPC;
$netconf = getnetconfig("udp");
```

Lorsque Perl détruit l'objet référencé par \$netconf, il envoie l'objet à la fonction XSUB DESTROY fournie. Perl ne peut pas déterminer, et ça ne l'intéresse pas, que cet objet est une structure C et non un objet Perl. En ce sens, il n'y a pas de différence entre l'objet créé par la XSUB getnetconfig() et un objet créé par une sous-routine Perl normale.

### 58.1.38 Le typemap

Le typemap est un ensemble de fragments de code utilisés par le compilateur **xsubpp** pour relier les paramètres de fonction et les valeurs C à des valeurs Perl. Le fichier typemap peut contenir trois sections dénommées TYPEMAP, INPUT et OUTPUT. La section INPUT indique au compilateur comment convertir des valeurs Perl en valeurs de certains types C. La section OUTPUT indique au compilateur comment traduire les valeurs de certains types C en des valeurs accessibles à Perl. La section TYPEMAP indique au compilateur quels fragments de code, dans les sections INPUT et OUTPUT, doivent être utilisés pour faire correspondre un type C donné à une valeur Perl. Chacune des sections du typemap doit être précédée de l'un des mots-clés TYPEMAP, INPUT et OUTPUT.

Le typemap par défaut, dans le répertoire `ext` du code source Perl, contient un grand nombre de types utiles qui sont mis à disposition des extensions Perl. Certaines extensions définissent des typemaps supplémentaires qu'elles peuvent conserver dans leur propre répertoire. Ces typemaps supplémentaires peuvent se référer aux tables de correspondance INPUT et OUTPUT du typemap principal. Le compilateur **xsubpp** permet au typemap d'une extension de redéfinir des correspondances présentes dans le typemap par défaut.

La plupart des extensions qui nécessitent un typemap personnalisé n'ont besoin que de la section TYPEMAP du fichier typemap. Le typemap personnalisé utilisé dans l'exemple `getnetconfig()` présenté précédemment montre ce qui est peut-être l'utilisation typique des typemaps dans les extensions. Ce typemap est utilisé pour faire correspondre une structure C au typemap `T_PTROBJ`. Le typemap utilisé par `getnetconfig()` est présenté ici. Remarquez que le type C est séparé du type XS avec une tabulation, et que l'opérateur unaire `*` de C est considéré comme une partie du nom du type C.

```
TYPEMAP
Netconfig *<tab>T_PTROBJ
```

Voici un exemple plus compliqué. Supposons que vous vouliez que `struct netconfig` soit béni dans la classe `Net::Config`. Une manière de le faire est d'utiliser de la manière suivante le caractère souligné (`_`) pour séparer les noms de paquetage :

```
typedef struct netconfig * Net_Config;
```

et de fournir ensuite une entrée de typemap `T_PTROBJ_SPECIAL` qui relie le souligné au quadruple point (`::`), puis de déclarer `Net_Config` avec ce type :

```
TYPEMAP
Net_Config T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
 if (sv_derived_from($arg, \"${(my $ntt=$ntype)=~s/_/:::/g;$ntt}\") {
 IV tmp = SvIV((SV*)SvRV($arg));
 $var = ($type) tmp;
 }
 else
 croak(\"$var n'est pas de type ${(my $ntt=$ntype)=~s/_/:::/g;$ntt}\")

OUTPUT
T_PTROBJ_SPECIAL
 sv_setref_pv($arg, \"${(my $ntt=$ntype)=~s/_/:::/g;$ntt}\",
 (void*)$var);
```

Les sections INPUT et OUTPUT substituent à la volée le souligné au quadruple point, ce qui produit l'effet désiré. Cet exemple donne une idée du pouvoir et de la souplesse du mécanisme des typemaps.

## 58.2 EXEMPLES

Fichier `RPC.xs` : interface avec certaines fonctions de la librairie de connexion ONC+ RPC.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <rpc/rpc.h>

typedef struct netconfig Netconfig;

MODULE = RPC PACKAGE = RPC

SV *
rpcb_gettime(host="localhost")
 char *host
 PREINIT:
 time_t timep;
 CODE:
 ST(0) = sv_newmortal();
 if(rpcb_gettime(host, &timep))
 sv_setnv(ST(0), (double)timep);

Netconfig *
getnetconfigent(netid="udp")
 char *netid

MODULE = RPC PACKAGE = NetconfigPtr PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
 Netconfig *netconf
 CODE:
 printf("NetconfigPtr::DESTROY\n");
 free(netconf);
```

Fichier `typemap` : typemap personnalisé pour `RPC.xs`.

```
TYPEMAP
Netconfig * T_PTROBJ
```

Fichier `RPC.pm` : module Perl pour l'extension RPC.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;
```

Fichier `rpctest.pl` : programme de test Perl pour l'extension RPC.

```
use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";
```



## 58.3 VERSION DE XS

Ce document couvre les fonctionnalités supportées par xsubpp 1.935.

## 58.4 AUTEUR

Dean Roehrich <roehrich@cray.com> Jul 8, 1996.

## 58.5 TRADUCTION

### 58.5.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005\_02. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 58.5.2 Traducteur

Thierry Bézecourt <thbz@worldnet.fr>

### 58.5.3 Relecture

Régis Julié <Regis.Julie@cetelem.fr>

# Chapitre 59

## perlintern

Documentation autogénérée de fonctions Perl purement internes

### 59.1 DESCRIPTION

Ce fichier est la documentation autogénérée de fonctions de l'interpréteur Perl qui sont documentées à l'aide du format interne de documentation de Perl mais qui ne sont pas marquées comme faisant partie de l'API de Perl. En d'autres termes, **elles ne doivent pas être utilisées dans des extensions !**

### 59.2 AUTEURS

Le système d'autodocumentation a été originellement ajouté au noyau Perl par Benjamin Stuhl. La documentation est de tous ceux qui ont été suffisamment bons pour documenter leurs fonctions.

### 59.3 VOIR AUSSI

*perlguts, perlapi*

### 59.4 TRADUCTION

#### 59.4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

#### 59.4.2 Traducteur

Roland Trique <[roland.trique@uhb.fr](mailto:roland.trique@uhb.fr)>

#### 59.4.3 Relecture

Gérard Delafond

# Chapitre 60

## perlapio

Interface d'abstraction des E/S internes à Perl

### 60.1 SYNOPSIS

```
PerlIO *PerlIO_stdin(void);
PerlIO *PerlIO_stdout(void);
PerlIO *PerlIO_stderr(void);

PerlIO *PerlIO_open(const char *,const char *);
int PerlIO_close(PerlIO *);

int PerlIO_stdoutf(const char *,...)
int PerlIO_puts(PerlIO *,const char *);
int PerlIO_putc(PerlIO *,int);
int PerlIO_write(PerlIO *,const void *,size_t);
int PerlIO_printf(PerlIO *, const char *,...);
int PerlIO_vprintf(PerlIO *, const char *, va_list);
int PerlIO_flush(PerlIO *);

int PerlIO_eof(PerlIO *);
int PerlIO_error(PerlIO *);
void PerlIO_clearerr(PerlIO *);

int PerlIO_getc(PerlIO *);
int PerlIO_ungetc(PerlIO *,int);
int PerlIO_read(PerlIO *,void *,size_t);

int PerlIO_fileno(PerlIO *);
PerlIO *PerlIO_fdopen(int, const char *);
PerlIO *PerlIO_importFILE(FILE *, int flags);
FILE *PerlIO_exportFILE(PerlIO *, int flags);
FILE *PerlIO_findFILE(PerlIO *);
void PerlIO_releaseFILE(PerlIO *,FILE *);

void PerlIO_setlinebuf(PerlIO *);

long PerlIO_tell(PerlIO *);
int PerlIO_seek(PerlIO *,off_t,int);
int PerlIO_getpos(PerlIO *,Fpos_t *);
int PerlIO_setpos(PerlIO *,Fpos_t *);
void PerlIO_rewind(PerlIO *);
```

```

int PerlIO_has_base(PerlIO *);
int PerlIO_has_cntptr(PerlIO *);
int PerlIO_fast_gets(PerlIO *);
int PerlIO_canset_cnt(PerlIO *);

char *PerlIO_get_ptr(PerlIO *);
int PerlIO_get_cnt(PerlIO *);
void PerlIO_set_cnt(PerlIO *,int);
void PerlIO_set_ptrcnt(PerlIO *,char *,int);
char *PerlIO_get_base(PerlIO *);
int PerlIO_get_bufsiz(PerlIO *);

```

## 60.2 DESCRIPTION

Les sources de Perl doivent utiliser les fonctions listées ci-dessus à la place de celles définies dans l'en-tête *stdio.h* de l'ANSI C. Les en-têtes de perl les remplacent par le mécanisme d'entrée-sortie sélectionné lors de l'exécution de Configure en utilisant des `#define`.

Ces fonctions sont calquées sur celles de *stdio.h*, mais pour certaines, l'ordre des paramètres a été réorganisé.

### PerlIO \*

Remplace FILE \*. Comme FILE \*, il doit être traité comme un type opaque (c'est probablement prudent de le considérer comme un pointeur sur quelque chose).

### PerlIO\_stdin(), PerlIO\_stdout(), PerlIO\_stderr()

À utiliser à la place de `stdin`, `stdout`, `stderr`. Elles ressemblent à des « appels de fonctions » plutôt qu'à des variables pour que ce soit plus facile d'en faire des appels de fonctions si la plate-forme ne peut exporter des données vers des modules chargés, ou si (par exemple) des « threads » différents peuvent avoir des valeurs différentes.

### PerlIO\_open(path, mode), PerlIO\_fdopen(fd,mode)

Correspondent à `fopen()/fdopen()` et utilise les mêmes arguments.

### PerlIO\_printf(fmt,...), PerlIO\_vprintf(f,fmt,a)

Équivalent à `fprintf()/vfprintf()`.

### PerlIO\_stdoutf(fmt,...)

Équivalent à `printf()`. `printf` est `#defined` à cette fonction, donc il est (pour le moment) légal d'utiliser `printf(fmt, ...)` dans les sources perl.

### PerlIO\_read(f,buf,compteur), PerlIO\_write(f,buf,compteur)

Correspondent à `fread()` et `fwrite()`. Attention, les arguments sont différents, Il y a seulement un « compteur » et le fichier `f` est en premier.

### PerlIO\_close(f)

### PerlIO\_puts(f,s), PerlIO\_putc(f,c)

Correspondent à `fputs()` et `fputc()`. Attention, les arguments ont été réorganisés pour que le fichier soit en premier.

### PerlIO\_ungetc(f,c)

Correspond à `ungetc()`. Attention, les arguments ont été réorganisé pour que le fichier soit en premier.

### PerlIO\_getc(f)

Correspond à `getc()`.

### PerlIO\_eof(f)

Correspond à `feof()`.

### PerlIO\_error(f)

Correspond à `ferror()`.

### PerlIO\_fileno(f)

Correspond à `fileno()`. Attention, sur certaines plates-formes, la définition de « `fileno` » peut ne pas correspondre à celle d'Unix.

### PerlIO\_clearerr(f)

Correspond à `clearerr()`, c.-à-d., retire les drapeaux « `eof` » et « `error` » pour le « flux ».

### PerlIO\_flush(f)

Correspond à `fflush()`.

**PerlIO\_tell(f)**

Correspond à ftell().

**PerlIO\_seek(f,o,w)**

Correspond à fseek().

**PerlIO\_getpos(f,p), PerlIO\_setpos(f,p)**

Correspondent à fgetpos() et fsetpos(). Si la plate-forme ne dispose pas de ces fonctions, elles sont implémentées à l'aide de PerlIO\_tell() et PerlIO\_seek().

**PerlIO\_rewind(f)**

Correspond à rewind(). Note : elle peut être définie à l'aide de PerlIO\_seek().

**PerlIO\_tmpfile()**

Correspond à tmpfile(), c.-à-d., retourne un PerlIO anonyme qui sera automatiquement effacé lors de sa fermeture.

**60.2.1 Co-existence avec stdio**

Il existe un support de la coexistence de PerlIO avec stdio. Évidemment, si PerlIO est implémenté à l'aide de stdio, il n'y a aucun problème. Mais si perlIO est implémenté au-dessus de sfio (par exemple) il doit exister des mécanismes permettant de créer un FILE \* qui peut être passé aux fonctions de bibliothèques qui utilisent stdio.

**PerlIO\_importFILE(f,flags)**

Permet d'obtenir un PerlIO \* à partir d'un FILE \*. Peut nécessiter plus d'arguments, interface en cours de réécriture.

**PerlIO\_exportFILE(f,flags)**

À partir d'un PerlIO \* renvoie un FILE \* pouvant être donné à du code devant être compilé et lié avec *stdio.h* de l'ANSI C.

Le fait qu'un FILE \* a été « exporté » est enregistré, et peut affecter de futures opérations sur le PerlIO \* original. PerlIO \*.

**PerlIO\_findFILE(f)**

Retourne un FILE \* prédéfini « exporté » (s'il existe). Ceci est un bouche-trou jusqu'à ce que l'interface soit complètement définie.

**PerlIO\_releaseFILE(p,f)**

L'appel à PerlIO\_releaseFILE informe PerlIO que le FILE \* ne sera plus utilisé. Il est alors retiré de la liste des FILE \* « exporté », et le PerlIO \* associé retrouve son comportement normal.

**PerlIO\_setlinebuf(f)**

Correspond à setlinebuf(). Son utilisation est dépréciée en attendant une décision sur son sort. (Le noyau de Perl l'utilise *uniquement* lors du dump ; cela n'a rien à voir avec le vidage automatique \$|.)

En plus de l'API utilisateur décrite précédemment, il existe une interface d'« implémentation » qui permet à Perl d'accéder aux structures internes de PerlIO. Les appels suivants correspondent aux diverses macros FILE\_XXX déterminées par Configure. Cette section n'a d'intérêt que pour ceux qui sont concernés par un descriptif détaillé de comportement du noyau perl ou qui implémentent un MAPPING PerlIO.

**PerlIO\_has\_cntptr(f)**

L'implémentation peut retourner un pointeur vers la position courante dans le « buffer » et le nombre d'octets disponibles dans le buffer.

**PerlIO\_get\_ptr(f)**

Retourne un pointeur vers le prochain octet du buffer à lire.

**PerlIO\_get\_cnt(f)**

Retourne le nombre d'octets du buffer restant à lire.

**PerlIO\_canset\_cnt(f)**

L'implémentation peut ajuster son idée sur le nombre d'octets dans le buffer.

**PerlIO\_fast\_gets(f)**

L'implémentation a toutes les interfaces nécessaires pour permettre au code rapide de perl d'utiliser le mécanisme <FILE> .

```
PerlIO_fast_gets(f) = PerlIO_has_cntptr(f) && \
 PerlIO_canset_cnt(f) && \
 'Can set pointer into buffer'
```

**PerlIO\_set\_ptrcnt(f,p,c)**

Place le pointeur dans le buffer et remplace le nombre d'octets encore dans le buffer. Doit être utilisé seulement pour positionner le pointeur à l'intérieur de la plage impliquée par un appel précédent à `PerlIO_get_ptr` et `PerlIO_get_cnt`.

**PerlIO\_set\_cnt(f,c)**

Obscure - force le nombre d'octets dans le buffer. Déprécié. Utilisé uniquement dans `doio.c` pour forcer un compteur `<-1` à `-1`. Deviendra peut-être `PerlIO_set_empty` ou similaire. Cet appel peut éventuellement ne rien faire si « compteur » est déduit d'un pointeur et d'une « limite ».

**PerlIO\_has\_base(f)**

L'implémentation a un buffer, et peut retourner un pointeur vers celui-ci ainsi que sa taille. Utilisé par `perl` pour les tests `-T` / `-B`. Les autres usages seraient très obscurs...

**PerlIO\_get\_base(f)**

Retourne le *début* du buffer.

**PerlIO\_get\_bufsiz(f)**

Retourne la *taille totale* du buffer.

## 60.3 TRADUCTION

### 60.3.1 Version

Cette traduction française correspond à la version anglaise distribuée avec `perl 5.005_02`. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 60.3.2 Traducteurs

Marc Carmier <carmier@immortels.frmug.org>

### 60.3.3 Relecture

Gérard Delafond.

## **Cinquième partie**

### **Divers**

# Chapitre 61

## perlbook

Livres concernant Perl

### 61.1 DESCRIPTION

Le Camel Book (livre du chameau), officiellement appelé *Programming Perl, Third Edition*, par Larry Wall et al, est la référence ultime en ce qui concerne Perl. Vous pouvez le commander ainsi que d'autres livres sur Perl chez O'Reilly & Associates, 1-800-998-9938 (aux usa). Pour l'international, +1 707 829 0515. Si vous pouvez récupérer un bon de commande O'Reilly, vous pouvez aussi leur faxer à +1 707 829 0104. Si vous êtes connecté au Web, vous pouvez même surfer sur <<http://www.oreilly.com/>> et trouver un bon de commande en ligne.

D'autres livres concernant Perl provenant de divers éditeurs et auteurs peuvent être trouvés dans *perlfqa2* ou sur le web <<http://books.perl.org>>.

### 61.2 TRADUCTION

#### 61.2.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

#### 61.2.2 Traducteur

Mathieu Arnold <[arn\\_mat@club-internet.fr](mailto:arn_mat@club-internet.fr)>. Paul Gaborit <[Paul.Gaborit@enstimac.fr](mailto:Paul.Gaborit@enstimac.fr)>.

#### 61.2.3 Relecture

Gérard Delafond



# Chapitre 62

## perlhist

Les archives de l'histoire de Perl

### 62.1 DESCRIPTION

Ce document vise à enregistrer les versions publiées du code source de Perl.

### 62.2 INTRODUCTION

L'histoire de Perl en bref, par Larry Wall:

```
Perl 0 introduction de Perl à mes collègues de bureau.
Perl 1 introduction de Perl au monde, et changement de
 /\(...\|\...\)/ en /(...\|...)/. \ (Dan Faigin ne
 me l'a toujours pas pardonné. :-\)
Perl 2 introduction du paquetage d'expressions régulières
 d'Henry Spencer.
Perl 3 introduction de la capacité à manipuler des données
 binaires (embedded nulls).
Perl 4 introduction du premier Camel book. Vraiment. Nous
 avons principalement juste changé les numéros de version
 pour que le livre puisse se référer à la 4.000.
Perl 5 introduction de tout le reste, y compris la capacité
 à introduire tout le reste.
```

### 62.3 LES GARDIENS DE LA CITROUILLE

Larry Wall, Andy Dougherty, Tom Christiansen, Charles Bailey, Nick Ing-Simmons, Chip Salzenberg, Tim Bunce, Malcolm Beattie, Gurusamy Sarathy, Graham Barr, Jarkko Hietaniemi, Hugo van der Sanden, Michael Schwern, Rafael Garcia-Suarez, Nicholas Clark, Richard Clamp, Leon Brocard.

#### 62.3.1 LA CITROUILLE ?

[extrait de Porting/pumpkin.pod dans la distribution du code source de Perl]

L'idée en revient à Chip Salzenberg, avec une pensée pour son co legs, David Croy. Nous nous étions passés différents noms (le bâton, le jeton, la patate chaude) mais aucun n'avait pris. Alors, Chip demanda :

[début de citation]

```
Who has the patch pumpkin?
(Qui a la citrouille de patch ?)
```

Pour expliquer : David Croy me dit un jour que dans un précédent emploi, il y avait un streamer utilisé par de multiples systèmes pour leurs sauvegardes. Mais au lieu d'un logiciel d'exclusion de haute technologie, ils utilisaient une méthode bas de gamme pour empêcher les sauvegardes multiples simultanées : a stuffed pumpkin. Personne n'avait la permission de faire une sauvegarde à moins d'avoir la "citrouille de sauvegarde" ("backup pumpkin").

[fin de citation]

Le nom est resté. Le possesseur de la citrouille est parfois appelé le pumpking ["roi de la citrouille", NDT] (gardant le source à flot ?) ou le pumpkineer [Jeu de mot avec "puppeteer", "marionnettiste", NDT] (tirant les ficelles ?).

## 62.4 LES ARCHIVES

Pump-king	Version	Date	Notes (en aucun cas complètes, voir Changes* pour plus de détails)
Larry	0	Classifiée.	Ne le demandez pas.
Larry	1.000	1987-Dec-18	
	1.001..10	1988-Jan-30	
	1.011..14	1988-Feb-02	
Schwern	1.0.15	2002-Dec-18	Modernisation
Richard	1.0.16	2003-Dec-18	
Larry	2.000	1988-Jun-05	
	2.001	1988-Jun-28	
Larry	3.000	1989-Oct-18	
	3.001	1989-Oct-26	
	3.002..4	1989-Nov-11	
	3.005	1989-Nov-18	
	3.006..8	1989-Dec-22	
	3.009..13	1990-Mar-02	
	3.014	1990-Mar-13	
	3.015	1990-Mar-14	
	3.016..18	1990-Mar-28	
	3.019..27	1990-Aug-10	Subroutines utilisateurs.
	3.028	1990-Aug-14	
	3.029..36	1990-Oct-17	
	3.037	1990-Oct-20	
	3.040	1990-Nov-10	
	3.041	1990-Nov-13	
	3.042..43	1991-Jan-??	
	3.044	1991-Jan-12	
Larry	4.000	1991-Mar-21	
	4.001..3	1991-Apr-12	
	4.004..9	1991-Jun-07	
	4.010	1991-Jun-10	
	4.011..18	1991-Nov-05	
	4.019	1991-Nov-11	Stable.
	4.020..33	1992-Jun-08	
	4.034	1992-Jun-11	
	4.035	1992-Jun-23	
Larry	4.036	1993-Feb-05	Très stable.

	5.000alpha1	1993-Jul-31	
	5.000alpha2	1993-Aug-16	
	5.000alpha3	1993-Oct-10	
	5.000alpha4	1993-???-??	
	5.000alpha5	1993-???-??	
	5.000alpha6	1994-Mar-18	
	5.000alpha7	1994-Mar-25	
Andy	5.000alpha8	1994-Apr-04	
Larry	5.000alpha9	1994-May-05	ext apparaît.
	5.000alpha10	1994-Jun-11	
	5.000alpha11	1994-Jul-01	
Andy	5.000a11a	1994-Jul-07	To fit 14.
	5.000a11b	1994-Jul-14	
	5.000a11c	1994-Jul-19	
	5.000a11d	1994-Jul-22	
Larry	5.000alpha12	1994-Aug-04	
Andy	5.000a12a	1994-Aug-08	
	5.000a12b	1994-Aug-15	
	5.000a12c	1994-Aug-22	
	5.000a12d	1994-Aug-22	
	5.000a12e	1994-Aug-22	
	5.000a12f	1994-Aug-24	
	5.000a12g	1994-Aug-24	
	5.000a12h	1994-Aug-24	
Larry	5.000beta1	1994-Aug-30	
Andy	5.000b1a	1994-Sep-06	
Larry	5.000beta2	1994-Sep-14	Noyau rendu gadouilleux.
Andy	5.000b2a	1994-Sep-14	
	5.000b2b	1994-Sep-17	
	5.000b2c	1994-Sep-17	
Larry	5.000beta3	1994-Sep-??	
Andy	5.000b3a	1994-Sep-18	
	5.000b3b	1994-Sep-22	
	5.000b3c	1994-Sep-23	
	5.000b3d	1994-Sep-27	
	5.000b3e	1994-Sep-28	
	5.000b3f	1994-Sep-30	
	5.000b3g	1994-Oct-04	
Andy	5.000b3h	1994-Oct-07	
Larry ?	5.000gamma	1994-Oct-13?	
Larry	5.000	1994-Oct-17	
Andy	5.000a	1994-Dec-19	
	5.000b	1995-Jan-18	
	5.000c	1995-Jan-18	
	5.000d	1995-Jan-18	
	5.000e	1995-Jan-18	
	5.000f	1995-Jan-18	
	5.000g	1995-Jan-18	
	5.000h	1995-Jan-18	
	5.000i	1995-Jan-26	
	5.000j	1995-Feb-07	
	5.000k	1995-Feb-11	
	5.000l	1995-Feb-21	
	5.000m	1995-Feb-28	
	5.000n	1995-Mar-07	
	5.000o	1995-Mar-13?	
Larry	5.001	1995-Mar-13	

Andy	5.001a	1995-Mar-15	
	5.001b	1995-Mar-31	
	5.001c	1995-Apr-07	
	5.001d	1995-Apr-14	
	5.001e	1995-Apr-18	Stable.
	5.001f	1995-May-31	
	5.001g	1995-May-25	
	5.001h	1995-May-25	
	5.001i	1995-May-30	
	5.001j	1995-Jun-05	
	5.001k	1995-Jun-06	
	5.001l	1995-Jun-06	Stable.
	5.001m	1995-Jul-02	Très stable.
	5.001n	1995-Oct-31	Très instable.
	5.002beta1	1995-Nov-21	
	5.002b1a	1995-Dec-04	
	5.002b1b	1995-Dec-04	
	5.002b1c	1995-Dec-04	
	5.002b1d	1995-Dec-04	
	5.002b1e	1995-Dec-08	
	5.002b1f	1995-Dec-08	
Tom	5.002b1g	1995-Dec-21	Doc publiée.
Andy	5.002b1h	1996-Jan-05	
	5.002b2	1996-Jan-14	
Larry	5.002b3	1996-Feb-02	
Andy	5.002gamma	1996-Feb-11	
Larry	5.002delta	1996-Feb-27	
Larry	5.002	1996-Feb-29	Prototypes.
Charles	5.002_01	1996-Mar-25	
	5.003	1996-Jun-25	Version sécurisée.
	5.003_01	1996-Jul-31	
Nick	5.003_02	1996-Aug-10	
Andy	5.003_03	1996-Aug-28	
	5.003_04	1996-Sep-02	
	5.003_05	1996-Sep-12	
	5.003_06	1996-Oct-07	
	5.003_07	1996-Oct-10	
Chip	5.003_08	1996-Nov-19	
	5.003_09	1996-Nov-26	
	5.003_10	1996-Nov-29	
	5.003_11	1996-Dec-06	
	5.003_12	1996-Dec-19	
	5.003_13	1996-Dec-20	
	5.003_14	1996-Dec-23	
	5.003_15	1996-Dec-23	
	5.003_16	1996-Dec-24	
	5.003_17	1996-Dec-27	
	5.003_18	1996-Dec-31	
	5.003_19	1997-Jan-04	
	5.003_20	1997-Jan-07	
	5.003_21	1997-Jan-15	
	5.003_22	1997-Jan-16	
	5.003_23	1997-Jan-25	
	5.003_24	1997-Jan-29	
	5.003_25	1997-Feb-04	
	5.003_26	1997-Feb-10	
	5.003_27	1997-Feb-18	

	5.003_28	1997-Feb-21	
	5.003_90	1997-Feb-25	On monte la rampe vers la version 5.004.
	5.003_91	1997-Mar-01	
	5.003_92	1997-Mar-06	
	5.003_93	1997-Mar-10	
	5.003_94	1997-Mar-22	
	5.003_95	1997-Mar-25	
	5.003_96	1997-Apr-01	
	5.003_97	1997-Apr-03	Assez largement utilisée.
	5.003_97a	1997-Apr-05	
	5.003_97b	1997-Apr-08	
	5.003_97c	1997-Apr-10	
	5.003_97d	1997-Apr-13	
	5.003_97e	1997-Apr-15	
	5.003_97f	1997-Apr-17	
	5.003_97g	1997-Apr-18	
	5.003_97h	1997-Apr-24	
	5.003_97i	1997-Apr-25	
	5.003_97j	1997-Apr-28	
	5.003_98	1997-Apr-30	
	5.003_99	1997-May-01	
	5.003_99a	1997-May-09	
	p54rc1	1997-May-12	Candidates à la publication.
	p54rc2	1997-May-14	
Chip	5.004	1997-May-15	Une version de maintenance majeure.
Tim	5.004_01-t1	1997-???-??	La série de maintenance 5.004.
	5.004_01-t2	1997-Jun-11	dit perl5.004m1t2
	5.004_01	1997-Jun-13	
	5.004_01_01	1997-Jul-29	dit perl5.004m2t1
	5.004_01_02	1997-Aug-01	dit perl5.004m2t2
	5.004_01_03	1997-Aug-05	dit perl5.004m2t3
	5.004_02	1997-Aug-07	
	5.004_02_01	1997-Aug-12	dit perl5.004m3t1
	5.004_03-t2	1997-Aug-13	dit perl5.004m3t2
	5.004_03	1997-Sep-05	
	5.004_04-t1	1997-Sep-19	dit perl5.004m4t1
	5.004_04-t2	1997-Sep-23	dit perl5.004m4t2
	5.004_04-t3	1997-Oct-10	dit perl5.004m4t3
	5.004_04-t4	1997-Oct-14	dit perl5.004m4t4
	5.004_04	1997-Oct-15	
	5.004_04-m1	1998-Mar-04	(5.004m5t1) Essais de maintenance pour 5.004_05.
	5.004_04-m2	1998-May-01	
	5.004_04-m3	1998-May-15	
	5.004_04-m4	1998-May-19	
	5.004_05-MT5	1998-Jul-21	
	5.004_05-MT6	1998-Oct-09	
	5.004_05-MT7	1998-Nov-22	
	5.004_05-MT8	1998-Dec-03	
Chip	5.004_05-MT9	1999-Apr-26	
	5.004_05	1999-Apr-29	
Malcolm	5.004_50	1997-Sep-09	La série de développement 5.005.
	5.004_51	1997-Oct-02	
	5.004_52	1997-Oct-15	
	5.004_53	1997-Oct-16	
	5.004_54	1997-Nov-14	
	5.004_55	1997-Nov-25	
	5.004_56	1997-Dec-18	
	5.004_57	1998-Feb-03	

	5.004_58	1998-Feb-06	
	5.004_59	1998-Feb-13	
	5.004_60	1998-Feb-20	
	5.004_61	1998-Feb-27	
	5.004_62	1998-Mar-06	
	5.004_63	1998-Mar-17	
	5.004_64	1998-Apr-03	
	5.004_65	1998-May-15	
	5.004_66	1998-May-29	
Sarathy	5.004_67	1998-Jun-15	
	5.004_68	1998-Jun-23	
	5.004_69	1998-Jun-29	
	5.004_70	1998-Jul-06	
	5.004_71	1998-Jul-09	
	5.004_72	1998-Jul-12	
	5.004_73	1998-Jul-13	
	5.004_74	1998-Jul-14	Candidate pour la 5.005 beta.
	5.004_75	1998-Jul-15	5.005 beta1.
	5.004_76	1998-Jul-21	5.005 beta2.
	5.005	1998-Jul-22	Oneperl.
Sarathy	5.005_01	1998-Jul-27	La série de maintenance 5.005.
	5.005_02-T1	1998-Aug-02	
	5.005_02-T2	1998-Aug-05	
	5.005_02	1998-Aug-08	
Graham	5.005_03-MT1	1998-Nov-30	
	5.005_03-MT2	1999-Jan-04	
	5.005_03-MT3	1999-Jan-17	
	5.005_03-MT4	1999-Jan-26	
	5.005_03-MT5	1999-Jan-28	
	5.005_03	1999-Mar-28	
Leon	5.005_04-RC1	2004-Feb-05	
	5.005_04-RC2	2004-Feb-18	
	5.005_04	2004-Feb-23	
Sarathy	5.005_50	1998-Jul-26	La série de développement 5.6.
	5.005_51	1998-Aug-10	
	5.005_52	1998-Sep-25	
	5.005_53	1998-Oct-31	
	5.005_54	1998-Nov-30	
	5.005_55	1999-Feb-16	
	5.005_56	1999-Mar-01	
	5.005_57	1999-May-25	
	5.005_58	1999-Jul-27	
	5.005_59	1999-Aug-02	
	5.005_60	1999-Aug-02	
	5.005_61	1999-Aug-20	
	5.005_62	1999-Oct-15	
	5.005_63	1999-Dec-09	
	5.5.640	2000-Feb-02	
	5.5.650	2000-Feb-08	beta1
	5.5.660	2000-Feb-22	beta2
	5.5.670	2000-Feb-29	beta3
	5.6.0-RC1	2000-Mar-09	Release candidate 1.
	5.6.0-RC2	2000-Mar-14	Release candidate 2.
	5.6.0-RC3	2000-Mar-21	Release candidate 3.
	5.6.0	2000-Mar-22	
Sarathy	5.6.1-TRIAL1	2000-Dec-18	La série de maintenance 5.6
	5.6.1-TRIAL2	2001-Jan-31	
	5.6.1-TRIAL3	2001-Mar-19	

	5.6.1-foolish	2001-Apr-01	La version "fools-gold".
	5.6.1	2001-Apr-08	
Rafael	5.6.2-RC1	2003-Nov-08	
	5.6.2	2003-Nov-15	Correctifs pour la nouvelle compilation
Jarkko	5.7.0	2000-Sep-02	La série 5.7: développement.
	5.7.1	2001-Apr-09	
	5.7.2	2001-Jul-13	Virtual release candidate 0.
	5.7.3	2002-Mar-05	
	5.8.0-RC1	2002-Jun-01	
	5.8.0-RC2	2002-Jun-21	
	5.8.0-RC3	2002-Jul-13	
	5.8.0	2002-Jul-18	
	5.8.1-RC1	2003-Jul-10	
	5.8.1-RC2	2003-Jul-11	
	5.8.1-RC3	2003-Jul-30	
	5.8.1-RC4	2003-Aug-01	
	5.8.1-RC5	2003-Sep-22	
	5.8.1	2003-Sep-25	
Nicholas	5.8.2-RC1	2003-Oct-27	
	5.8.2-RC2	2003-Nov-03	
	5.8.2	2003-Nov-05	
	5.8.3-RC1	2004-Jan-07	
	5.8.3	2004-Jan-14	
	5.8.4-RC1	2004-Apr-05	
	5.8.4-RC2	2004-Apr-15	
	5.8.4	2004-Apr-21	
	5.8.5-RC1	2004-Jul-06	
	5.8.5-RC2	2004-Jul-08	
	5.8.5	2004-Jul-19	
	5.8.6-RC1	2004-Nov-11	
	5.8.6	2004-Nov-27	
	5.8.7-RC1	2005-May-18	
	5.8.7	2005-May-30	
	5.8.8-RC1	2006-Jan-20	
	5.8.8	2006-Jan-31	
Hugo	5.9.0	2003-Oct-27	
Rafael	5.9.1	2004-Mar-16	
	5.9.2	2005-Apr-01	
	5.9.3	2006-Jan-28	
	5.9.4	2006-Aug-15	
	5.9.5	2007-Jul-07	
	5.10.0-RC1	2007-Nov-17	
	5.10.0-RC2	2007-Nov-25	
	5.10.0	2007-Dec-18	

### 62.4.1 TAILLES DE VERSIONS CHOISIES

Par exemple, la notation "core: 212 29" pour la version 1.000 signifie qu'elle contenait 212 kilo-octets dans le noyau, en 29 fichiers. Les colonnes "core".. "doc" sont expliquées ci-dessous.

version	core		lib		ext		t		doc	
=====	=====	=====	=====	=====	=====	=====	=====	=====	=====	=====
1.000	212	29	-	-	-	-	38	51	62	3
1.014	219	29	-	-	-	-	39	52	68	4
2.000	309	31	2	3	-	-	55	57	92	4
2.001	312	31	2	3	-	-	55	57	94	4
3.000	508	36	24	11	-	-	79	73	156	5

3.044	645	37	61	20	-	-	90	74	190	6
4.000	635	37	59	20	-	-	91	75	198	4
4.019	680	37	85	29	-	-	98	76	199	4
4.036	709	37	89	30	-	-	98	76	208	5
5.000alpha2	785	50	114	32	-	-	112	86	209	5
5.000alpha3	801	50	117	33	-	-	121	87	209	5
5.000alpha9	1022	56	149	43	116	29	125	90	217	6
5.000a12h	978	49	140	49	205	46	152	97	228	9
5.000b3h	1035	53	232	70	216	38	162	94	218	21
5.000	1038	53	250	76	216	38	154	92	536	62
5.001m	1071	54	388	82	240	38	159	95	544	29
5.002	1121	54	661	101	287	43	155	94	847	35
5.003	1129	54	680	102	291	43	166	100	853	35
5.003_07	1231	60	748	106	396	53	213	137	976	39
5.004	1351	60	1230	136	408	51	355	161	1587	55
5.004_01	1356	60	1258	138	410	51	358	161	1587	55
5.004_04	1375	60	1294	139	413	51	394	162	1629	55
5.004_05	1463	60	1435	150	394	50	445	175	1855	59
5.004_51	1401	61	1260	140	413	53	358	162	1594	56
5.004_53	1422	62	1295	141	438	70	394	162	1637	56
5.004_56	1501	66	1301	140	447	74	408	165	1648	57
5.004_59	1555	72	1317	142	448	74	424	171	1678	58
5.004_62	1602	77	1327	144	629	92	428	173	1674	58
5.004_65	1626	77	1358	146	615	92	446	179	1698	60
5.004_68	1856	74	1382	152	619	92	463	187	1784	60
5.004_70	1863	75	1456	154	675	92	494	194	1809	60
5.004_73	1874	76	1467	152	762	102	506	196	1883	61
5.004_75	1877	76	1467	152	770	103	508	196	1896	62
5.005	1896	76	1469	152	795	103	509	197	1945	63
5.005_03	1936	77	1541	153	813	104	551	201	2176	72
5.005_50	1969	78	1842	301	795	103	514	198	1948	63
5.005_53	1999	79	1885	303	806	104	602	224	2002	67
5.005_56	2086	79	1970	307	866	113	672	238	2221	75
5.6.0	2930	80	2626	364	1096	129	868	281	2841	93
5.7.0	2977	80	2801	425	1250	132	975	307	3206	100
5.6.1	3049	80	3764	484	1924	159	1025	304	3593	119
5.7.1	3351	84	3442	455	1944	167	1334	357	3698	124
5.7.2	3491	87	4858	618	3290	298	1598	449	3910	139
5.7.3	3415	87	5367	630	14448	410	2205	640	4491	148

Les colonnes "core"... "doc" désignent les fichiers suivants dans la distribution du code source de Perl. La notation globalisée \*\* signifie récursivement, (.) désigne les fichiers ordinaires.

```

core *. [hcy]
lib lib/**/* .p[ml]
ext ext/**/*.{[hcyt],xs,pm}
t t/**/*(.) (pour 1-5.005_56) ou **/* .t (pour 5.6.0-5.7.3)
doc {README*,INSTALL,*[_.]man{,?.},pod/**/* .pod}

```

Voici des statistiques pour les autres sous-répertoires et pour un fichier dans la distribution du source de Perl, pour des versions un peu plus sélectionnées.

```

=====
Légende : ko #

 1.014 2.001 3.044 4.000 4.019 4.036

atarist - - - - - - - 113 31
Configure 31 1 37 1 62 1 73 1 83 1 86 1
eg - - 34 28 47 39 47 39 47 39 47 39

```



emacs	-	-	-	-	-	67	4	67	4	67	4
h2pl	-	-	-	-	12	12	12	12	12	12	12
hints	-	-	-	-	-	-	-	5	42	11	56
msdos	-	-	-	-	41	13	57	15	58	15	60
os2	-	-	-	-	63	22	81	29	81	29	113
usub	-	-	-	-	21	16	25	7	43	8	43
x2p	103	17	104	17	137	17	147	18	152	19	154

=====

	5.000a2	5.000a12h	5.000b3h	5.000	5.001m	5.002	5.003
--	---------	-----------	----------	-------	--------	-------	-------

atarist	113	31	113	31	-	-	-	-	-	-	-
bench	-	-	0	1	-	-	-	-	-	-	-
Bugs	2	5	26	1	-	-	-	-	-	-	-
dlperl	40	5	-	-	-	-	-	-	-	-	-
do	127	71	-	-	-	-	-	-	-	-	-
Configure	-	-	153	1	159	1	160	1	180	1	201
Doc	-	-	26	1	75	7	11	1	11	1	-
eg	79	58	53	44	51	43	54	44	54	44	54
emacs	67	4	104	6	104	6	104	1	104	6	108
h2pl	12	12	12	12	12	12	12	12	12	12	12
hints	11	56	12	46	18	48	18	48	44	56	73
msdos	60	15	60	15	-	-	-	-	-	-	-
os2	113	31	113	31	-	-	-	-	84	17	56
U	-	-	62	8	112	42	-	-	-	-	-
usub	43	8	-	-	-	-	-	-	-	-	-
utils	-	-	-	-	-	-	-	-	87	7	88
vms	-	-	80	7	123	9	184	15	304	20	500
x2p	171	22	171	21	162	20	162	20	279	20	280

=====

	5.003_07	5.004	5.004_04	5.004_62	5.004_65	5.004_68
--	----------	-------	----------	----------	----------	----------

beos	-	-	-	-	-	1	1	1	1
Configure	217	1	225	1	225	1	240	1	248
cygwin32	-	-	23	5	23	5	23	5	24
djgpp	-	-	-	-	-	14	5	14	5
eg	54	44	81	62	81	62	81	62	81
emacs	143	1	194	1	204	1	212	2	212
h2pl	12	12	12	12	12	12	12	12	12
hints	90	62	129	69	132	71	144	72	151
os2	117	42	121	42	127	42	127	44	129
plan9	79	15	82	15	82	15	82	15	82
Porting	51	1	94	2	109	4	203	6	234
qnx	-	-	1	2	1	2	1	2	1
utils	97	7	112	8	118	8	124	8	156
vms	505	27	518	34	524	34	538	34	569
win32	-	-	285	33	378	36	470	39	493
x2p	280	19	281	19	281	19	281	19	282

=====

	5.004_70	5.004_73	5.004_75	5.005	5.005_03
--	----------	----------	----------	-------	----------

apollo	-	-	-	-	-	0	1
beos	1	1	1	1	1	1	1
Configure	256	1	256	1	264	1	264
cygwin32	24	5	24	5	24	5	24

djgpp	14	5	14	5	14	5	14	5	15	5
eg	86	65	86	65	86	65	86	65	86	65
emacs	262	2	262	2	262	2	262	2	274	2
h2pl	12	12	12	12	12	12	12	12	12	12
hints	157	74	157	74	159	74	160	74	179	77
mint	-	-	-	-	-	-	-	-	4	7
mpeix	-	-	-	-	5	3	5	3	5	3
os2	129	44	139	44	142	44	143	44	148	44
plan9	82	15	82	15	82	15	82	15	82	15
Porting	241	9	253	9	259	10	264	12	272	13
qnx	1	2	1	2	1	2	1	2	1	2
utils	160	9	160	9	160	9	160	9	164	9
vms	570	34	572	34	573	34	575	34	583	34
vos	-	-	-	-	-	-	-	-	156	10
win32	577	41	585	41	585	41	587	41	600	42
x2p	281	19	281	19	281	19	281	19	281	19

### 62.4.2 TAILLES DE PATCH CHOISIES

Les colonnes "diff lignes ko" signifient que par exemple le patch 5.003\_08, qui doit être appliqué au-dessus de la 5.003\_07 (ou de ce qui précédait la 5.003\_08) ajoutait des lignes pour un volume de 110 kilo-octets, en retirait pour un volume de 19 ko, et en changeait pour 424 ko. Les lignes elles-mêmes uniquement sont comptées, par leur contexte. Le "+ - !" provient du format de sortie de diff(1).

Pump- king	Version	Date	diff lignes ko		
			+	-	!
=====					
Chip	5.003_08	1996-Nov-19	110	19	424
	5.003_09	1996-Nov-26	38	9	248
	5.003_10	1996-Nov-29	29	2	27
	5.003_11	1996-Dec-06	73	12	165
	5.003_12	1996-Dec-19	275	6	436
	5.003_13	1996-Dec-20	95	1	56
	5.003_14	1996-Dec-23	23	7	333
	5.003_15	1996-Dec-23	0	0	1
	5.003_16	1996-Dec-24	12	3	50
	5.003_17	1996-Dec-27	19	1	14
	5.003_18	1996-Dec-31	21	1	32
	5.003_19	1997-Jan-04	80	3	85
	5.003_20	1997-Jan-07	18	1	146
	5.003_21	1997-Jan-15	38	10	221
	5.003_22	1997-Jan-16	4	0	18
	5.003_23	1997-Jan-25	71	15	119
	5.003_24	1997-Jan-29	426	1	20
	5.003_25	1997-Feb-04	21	8	169
	5.003_26	1997-Feb-10	16	1	15
	5.003_27	1997-Feb-18	32	10	38
	5.003_28	1997-Feb-21	58	4	66
	5.003_90	1997-Feb-25	22	2	34
	5.003_91	1997-Mar-01	37	1	39
	5.003_92	1997-Mar-06	16	3	69
	5.003_93	1997-Mar-10	12	3	15
	5.003_94	1997-Mar-22	407	7	200
	5.003_95	1997-Mar-25	41	1	37
	5.003_96	1997-Apr-01	283	5	261
	5.003_97	1997-Apr-03	13	2	34
	5.003_97a	1997-Apr-05	57	1	27
	5.003_97b	1997-Apr-08	14	1	20
	5.003_97c	1997-Apr-10	20	1	16

	5.003_97d	1997-Apr-13	8	0	16
	5.003_97e	1997-Apr-15	15	4	46
	5.003_97f	1997-Apr-17	7	1	33
	5.003_97g	1997-Apr-18	6	1	42
	5.003_97h	1997-Apr-24	23	3	68
	5.003_97i	1997-Apr-25	23	1	31
	5.003_97j	1997-Apr-28	36	1	49
	5.003_98	1997-Apr-30	171	12	539
	5.003_99	1997-May-01	6	0	7
	5.003_99a	1997-May-09	36	2	61
	p54rc1	1997-May-12	8	1	11
	p54rc2	1997-May-14	6	0	40
	5.004	1997-May-15	4	0	4
Tim	5.004_01	1997-Jun-13	222	14	57
	5.004_02	1997-Aug-07	112	16	119
	5.004_03	1997-Sep-05	109	0	17
	5.004_04	1997-Oct-15	66	8	173

## 62.5 LES GARDIENS DES ARCHIVES

Jarkko Hietaniemi <[jhi@iki.fi](mailto:jhi@iki.fi)>.

Merci à la mémoire collective des Perliens. Outre les Gardiens de la Citrouille, Alan Champion, Mark Dominus, Andreas König, John Macdonald, Matthias Neeracher, Jeff Okamoto, Michael Pepler, Randal Schwartz et Paul D. Smith ont envoyé des corrections et des additions.

## 62.6 TRADUCTION

### 62.6.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

### 62.6.2 Traducteur

Roland Trique <[roland.trique@free.fr](mailto:roland.trique@free.fr)>. Maj vers 5.8.5, 5.8.8 puis 5.10.0 : Paul Gaborit (Paul.Gaborit @ enstimac.fr).

### 62.6.3 Relecture

Régis Julié <[regis.julie@cetelem.fr](mailto:regis.julie@cetelem.fr)>, Gérard Delafond.

# Chapitre 63

## perlartistic

La licence « artistique » Perl

### 63.1 SYNOPSIS

Vous pouvez faire référence à ce document en Pod via "L<perlartistic>"  
Vous pouvez aussi voir ce document en tapant la commande "perldoc perlartistic"

### 63.2 DESCRIPTION

Voici la **licence « artistique »**. Elle est ici pour que les modules, programmes, etc. qui veulent l'utiliser comme licence de distribution puissent s'y référer.

C'est aussi l'une des deux licences sous lesquelles Perl peut être redistribuer et/ou modifier. Pour voir la seconde, la Licence Publique Générale GNU, lisez *perlgpl*.

### 63.3 Avertissement du traducteur

Je suis déjà bien en peine de comprendre si l'une quelconque de la pléthore des licences de logiciel libre, en version originale, a la moindre valeur juridique, en a déjà eu, en aura jamais, a jamais été présentée avec succès au cours d'un procès... La présente traduction n'a d'autre vue que de faciliter l'appréhension par le lecteur francophone mal à l'aise en anglais des idées du texte original, et en aucun cas ne prétend avoir quoi que ce soit d'officiel, d'avalisé, ni de légal.

La version originale en anglais suit la traduction.

### 63.4 La « licence artistique »

#### 63.4.1 Préambule

Ce document a pour but de définir les conditions sous lesquelles un paquetage peut être copié, de sorte que le détenteur du copyright puisse garder une quelconque apparence de contrôle artistique sur le développement du paquetage, tout en donnant aux utilisateurs du paquetage le droit de l'utiliser et de le distribuer selon des usages plus ou moins établis, ainsi que le droit d'opérer des modifications raisonnables.

#### 63.4.2 Définitions

« **paquetage** »

se réfère à la collection de fichiers distribués par le détenteur du copyright, et ses dérivés créés à travers des modifications sur le texte.

« **version standard** »

se réfère à un tel paquetage s'il n'a pas été modifié, ou s'il l'a été en accord avec les souhaits du détenteur du copyright, comme spécifié ci-dessous.

**le « détenteur du copyright »**

est toute personne nommément citée dans le copyright ou qui possède un copyright sur le paquetage.

**« vous »**

c'est vous, si vous envisagez de copier ou de distribuer ce paquetage.

**un « coût de copie raisonnable »**

peut-on le justifier en fonction du prix des supports, des coûts de duplication, du temps consacré à cela, etc (ce n'est pas au détenteur du copyright qu'il vous faudra justifier ce prix, mais à la communauté des utilisateurs d'ordinateurs sans son ensemble, en tant que marché qui doit s'acquitter de cette somme).

**« disponible gratuitement »**

signifie qu'on ne demande rien pour l'objet en lui-même, mais que la manipulation de l'objet peut induire divers coûts. Cela signifie aussi que les détenteurs de l'objet ont la liberté de le redistribuer selon les mêmes conditions que celles qui l'accompagnaient quand ils l'ont reçu.

**63.4.3 Conditions de modification et de distribution**

1. Vous pouvez faire et donner des copies verbatim de la version standard de ce paquetage sous forme de source sans restriction, pourvu que vous dupliquiez toutes les notices de copyright originales et les mises en garde qui les accompagnent.
2. Vous pouvez appliquer des corrections de bogues, des corrections de portabilité, et d'autres modifications dérivées du domaine public ou du détenteur du copyright. Un paquetage modifié ainsi sera encore considéré comme étant la version standard.
3. Vous pouvez modifier votre copie de ce paquetage de toute autre manière, pourvu que vous insériez et mettiez en évidence une notice dans chacun des fichiers modifiés exposant comment et quand vous avez modifié ce fichier, et pourvu que vous fassiez au moins une chose parmi les suivantes :
  - a) placer vos modifications dans le domaine public ou les rendre disponibles gratuitement, par exemple en les publiant sur l'Usenet ou sur un médium équivalent, ou en plaçant ces modifications sur un site d'archives majeur tel que uunet.uu.net, ou en autorisant le détenteur du copyright à inclure vos modifications dans la versions standard du paquetage.
  - b) n'utiliser le paquetage ainsi modifié qu'au sein de votre société ou de votre organisation.
  - c) changer le nom de tous les exécutables non standard de sorte qu'ils n'entrent pas en conflit avec les exécutables habituels, qui devront eux aussi être proposés, et fournir une page de manuel séparée pour chaque exécutable non standard documentant clairement en quoi il diffère de la version standard.
  - d) négocier avec le détenteur du copyright un accord pour la distribution.
4. Vous pouvez distribuer les programmes de ce paquetage sous forme de code objet ou exécutable, pourvu que vous fassiez au moins une chose parmi les suivantes :
  - a) distribuer une version standard des exécutables et des fichiers de bibliothèques, ainsi que les instructions (dans la page de manuel ou son équivalent) sur l'endroit où trouver la version standard.
  - b) accompagner la distribution du source du paquetage, sous une forme lisible par la machine, avec vos modifications.
  - c) donner à tout exécutable non standard un nom non standard, et documenter clairement les différences dans les pages de manuel (ou leur équivalent), ainsi que les instructions sur l'endroit où obtenir la version standard.
  - d) procéder à tout autre accord pour la distribution avec le détenteur du copyright.
5. Vous pouvez demander un coût de copie raisonnable pour toute distribution de ce paquetage. Vous pouvez demander le prix de votre choix pour assurer l'assistance technique de ce paquetage. Vous ne pouvez pas exiger d'argent pour ce paquetage en lui-même. Cependant, vous pouvez distribuer ce paquetage dans un agrégat, aux côtés d'autres

programmes (éventuellement commerciaux), en tant qu'élément d'une distribution logicielle plus importante (éventuellement commerciale), pourvu que vous ne fassiez pas la publicité de ce paquetage comme étant de votre fait. Vous pouvez enchâsser l'interpréteur de ce paquetage dans un de vos exécutables (grâce à des liens); cela sera considéré comme une simple forme d'agrégation, pourvu que la version standard complète de l'interpréteur soit ainsi enchâssée.

6. Les scripts et les fichiers de bibliothèques fournis en entrée à ou produits en sortie des programmes de ce paquetage ne tombent pas automatiquement sous le copyright de ce paquetage, mais appartiennent à qui les a engendrés, et peuvent être vendus, ainsi qu'agrégés avec ce paquetage. Si de tels scripts ou fichiers de bibliothèques sont agrégés avec ce paquetage à travers les prétendues méthodes undump ou unexec de production d'une image binaire exécutable, alors la distribution d'une telle image ne sera pas considérée comme une distribution de ce paquetage, et ne tombera pas non plus sous le coup des restrictions des paragraphes 3 et 4, pourvu que vous ne représentiez une telle image exécutable comme une version standard de ce paquetage.
7. Les sous-programmes en C (ou, de manière comparable, les sous-programmes compilés dans d'autres langages) fournis par vous et liés avec ce paquetage dans le but d'émuler les sous-routines et les variables du langage défini par ce paquetage ne seront pas considérés comme faisant partie de ce paquetage, mais sont les équivalents de données en entrée comme il en est question dans le paragraphe 6, pourvu que ces sous-programmes ne modifient le langage d'aucune manière qui lui fasse échouer les tests de non régression pour le langage.
8. Il est toujours autorisé d'agréger ce paquetage avec une distribution commerciale pourvu que l'utilisation en soit enchâssée; c'est-à-dire sans aucune tentative manifeste de rendre les interfaces de ce paquetage visibles à l'utilisateur final de la distribution commerciale. Une telle utilisation ne sera pas considérée comme une distribution de ce paquetage.
9. Le nom du détenteur du copyright ne sera pas utilisé pour signer ou promouvoir des produits dérivés de ce logiciel sans autorisation écrite spécifique préalable.
10. CE PAQUETAGE EST FOURNI « EN L'ÉTAT » ET SANS GARANTIE SPÉCIFIQUE OU IMPLICITE, Y COMPRIS, ET C'EST UNE LISTE NON EXHAUSTIVE, LES GARANTIES IMPLICITES DE QUALITÉ COMMERCIALE OU D'ADÉQUATION À UN OBJECTIF PARTICULIER.

Fin.

## 63.5 TRADUCTION

### 63.5.1 Version

Cette traduction française correspond à la version anglaise (<http://www.perl.com/pub/language/misc/Artistic.html>). C'est aussi celle distribuée avec la version perl 5.10.0.

### 63.5.2 Traducteur

Sébastien Blondeel (sbi@linux-france.org). Conversion en Pod par Paul Gaborit (Paul.Gaborit @ enstimac.fr).

### 63.5.3 Relecture

Personne pour l'instant.

## 63.6 The "Artistic License"

### 63.6.1 Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

### 63.6.2 Definitions

**"Package"**

refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

**"Standard Version"**

refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

**"Copyright Holder"**

is whoever is named in the copyright or copyrights for the package.

**"You"**

is you, if you're thinking about copying or distributing this Package.

**"Reasonable copying fee"**

is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

**"Freely Available"**

means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

### 63.6.3 Conditions

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
  - a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
  - b) use the modified Package only within your corporation or organization.
  - c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
  - d) make other distribution arrangements with the Copyright Holder.
4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
  - a) distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
  - b) accompany the distribution with the machine-readable source of the Package with your modifications.
  - c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
  - d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whoever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.
7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.
8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.
9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End



## **Sixième partie**

# **Spécificités pour certaines langues**

## **Septième partie**

# **Spécificités pour certaines plates-formes**

# Table des matières

<b>1</b>	<b>Avant-propos</b>	<b>1</b>
<b>I</b>	<b>Présentation</b>	<b>2</b>
<b>2</b>	<b>perl</b>	<b>3</b>
2.1	SYNOPSIS	3
2.2	DESCRIPTION	3
2.2.1	Présentation	3
2.2.2	Tutoriels	4
2.2.3	Manuel de référence	4
2.2.4	Implémentation et interface avec le langage C	5
2.2.5	Divers	5
2.2.6	Spécificités pour certaines langues	6
2.2.7	Spécificités pour certaines plateformes	6
2.3	DISPONIBILITÉ	7
2.4	ENVIRONNEMENT	7
2.5	AUTEUR	8
2.6	FICHIERS	8
2.7	VOIR AUSSI	8
2.8	DIAGNOSTICS	8
2.9	BUGS	8
2.10	NOTES	9
2.11	TRADUCTION	9
2.11.1	Version	9
2.11.2	Traducteur	9
2.11.3	Relecture	9
<b>3</b>	<b>perlintro</b>	<b>10</b>
3.1	DESCRIPTION	10
3.1.1	Qu'est-ce que Perl ?	10
3.1.2	Exécuter des programmes Perl	10
3.1.3	Filet de sécurité	11
3.1.4	Les bases de la syntaxe	11
3.1.5	Types de variables Perl	12
3.1.6	Portée des variables	14
3.1.7	Structures conditionnelles et boucles	14
3.1.8	Opérateurs et fonctions internes	15
3.1.9	Fichiers et E/S (entrées/sorties)	16

3.1.10	Expressions régulières (ou rationnelles)	17
3.1.11	Écriture de sous-programmes	18
3.1.12	Perl orienté objet	19
3.1.13	Utilisations de modules Perl	19
3.2	AUTEUR	19
3.3	TRADUCTION	19
3.3.1	Version	19
3.3.2	Traducteur	19
3.3.3	Relecture	19
<b>II</b>	<b>Tutoriels</b>	<b>20</b>
<b>4</b>	<b>perlreftut</b>	<b>21</b>
4.1	DESCRIPTION	21
4.2	Qui a besoin de structures de données compliquées ?	21
4.3	La solution	22
4.4	Syntaxe	22
4.4.1	Construire des références	22
4.4.2	Utiliser les références	23
4.5	Un exemple	24
4.6	Règle de la flèche	25
4.7	Solution	25
4.8	Le reste	26
4.9	Résumé	27
4.10	Crédits	27
4.10.1	Distribution conditions	27
4.10.2	Conditions de distribution	27
4.11	TRADUCTION	28
4.11.1	Version	28
4.11.2	Traducteur	28
4.11.3	Relecture	28
<b>5</b>	<b>perldsc</b>	<b>29</b>
5.1	DESCRIPTION	29
5.2	RÉFÉRENCES	30
5.3	ERREURS COURANTES	30
5.4	AVERTISSEMENT SUR LA PRÉCÉDENCE	32
5.5	POURQUOI TOUJOURS UTILISER <code>use strict</code>	32
5.6	DÉBOGAGE	33
5.7	EXEMPLES DE CODE	33
5.8	TABLEAUX DE TABLEAUX	33
5.8.1	Déclaration d'un TABLEAU DE TABLEAUX	33
5.8.2	Génération d'un TABLEAU DE TABLEAUX	33
5.8.3	Accès et affichage d'un TABLEAU DE TABLEAUX	34
5.9	HACHAGE DE TABLEAUX	34
5.9.1	Déclaration d'un HACHAGE DE TABLEAUX	34
5.9.2	Génération d'un HACHAGE DE TABLEAUX	34

5.9.3	Accès et affichage d'un HACHAGE DE TABLEAUX	35
5.10	TABLEAUX DE HACHAGES	35
5.10.1	Déclaration d'un TABLEAU DE HACHAGES	35
5.10.2	Génération d'un TABLEAU DE HACHAGES	36
5.10.3	Accès et affichage d'un TABLEAU DE HACHAGES	36
5.11	HACHAGES DE HACHAGES	37
5.11.1	Déclaration d'un HACHAGE DE HACHAGES	37
5.11.2	Génération d'un HACHAGE DE HACHAGES	37
5.11.3	Accès et affichage d'un HACHAGE DE HACHAGES	38
5.12	ENREGISTREMENTS PLUS ÉLABORÉS	39
5.12.1	Déclaration d'ENREGISTREMENTS PLUS ÉLABORÉS	39
5.12.2	Déclaration d'un HACHAGE D'ENREGISTREMENTS COMPLEXES	39
5.12.3	Génération d'un HACHAGE D'ENREGISTREMENTS COMPLEXES	40
5.13	Liens avec les bases de données	41
5.14	VOIR AUSSI	41
5.15	AUTEUR	41
5.16	TRADUCTION	41
5.16.1	Version	41
5.16.2	Traducteur	42
5.16.3	Relecture	42
<b>6</b>	<b>perlol</b>	<b>43</b>
6.1	DESCRIPTION	43
6.1.1	Déclaration et accès aux tableaux de tableaux	43
6.1.2	Développer la vôtre	44
6.1.3	Accès et sortie	46
6.1.4	Tranches	47
6.2	VOIR AUSSI	47
6.3	AUTEUR	48
6.4	TRADUCTION	48
6.4.1	Version	48
6.4.2	Traducteur	48
6.4.3	Relecture	48
<b>7</b>	<b>perlrequick</b>	<b>49</b>
7.1	DESCRIPTION	49
7.2	Le guide	49
7.2.1	Reconnaissance de mot simple	49
7.2.2	Utilisation des classes de caractères	51
7.2.3	Reconnaître ceci ou cela	52
7.2.4	Groupement et hiérarchie	52
7.2.5	Mémorisation de la correspondance	53
7.2.6	Répétitions et quantificateurs	53
7.2.7	Plus de correspondances	54
7.2.8	Recherche et remplacement	54
7.2.9	L'opérateur de découpage : split	55
7.3	BUGS	55
7.4	VOIR AUSSI	55
7.5	AUTEUR ET COPYRIGHT	56
7.5.1	Remerciements	56
7.6	TRADUCTION	56
7.6.1	Version	56
7.6.2	Traducteur	56
7.6.3	Relecture	56

<b>8 perlretut</b>	<b>57</b>
8.1 DESCRIPTION	57
8.2 Partie 1: les bases	57
8.2.1 Reconnaissance d'un mot simple	57
8.2.2 Utilisation des classes de caractères	61
8.2.3 Reconnaître ceci ou cela	64
8.2.4 Regroupement et reconnaissance hiérarchique	64
8.2.5 Extraire ce qui est reconnu	65
8.2.6 Les références arrières	66
8.2.7 Références arrières relatives	66
8.2.8 Références arrières nommées	67
8.2.9 Numérotation des groupes dans une alternative	67
8.2.10 Information de position	67
8.2.11 Regroupements sans mémorisation	68
8.2.12 Reconnaissances répétées	68
8.2.13 Quantificateurs possessifs	72
8.2.14 Construction d'une expression rationnelle	73
8.2.15 Utilisation des expressions rationnelles en Perl	74
8.3 Partie 2: au-delà	79
8.3.1 Les plus des caractères, des chaînes et des classes de caractères	79
8.3.2 Compilation et stockage d'expressions rationnelles	81
8.3.3 Composition d'expressions rationnelles durant l'exécution	82
8.3.4 Commentaires et modificateurs intégrés dans une expression rationnelle	83
8.3.5 Regarder en arrière et regarder en avant	84
8.3.6 Utilisation de sous-expressions indépendantes pour empêcher les retours arrière	85
8.3.7 Les expressions conditionnelles	86
8.3.8 Définir des motifs nommés	87
8.3.9 Motifs récursifs	87
8.3.10 Un peu de magie : exécution de code Perl dans une expression rationnelle	87
8.3.11 Ordres de contrôle du retour-arrière	91
8.3.12 Directives (pragma) et déverminage	91
8.4 BUGS	93
8.5 VOIR AUSSI	93
8.6 AUTEURS ET COPYRIGHT	93
8.6.1 Remerciements	93
8.7 TRADUCTION	94
8.7.1 Version	94
8.7.2 Traducteur	94
8.7.3 Relecture	94

<b>9</b>	<b>perlboot</b>	<b>95</b>
9.1	DESCRIPTION	95
9.1.1	Si nous pouvions parler aux animaux...	95
9.1.2	Présentation de l'appel de méthodes via l'opérateur flèche	96
9.1.3	Et pour tout un troupeau	96
9.1.4	Le paramètre implicite de l'appel de méthodes	97
9.1.5	Appel à une seconde méthode pour simplifier les choses	97
9.1.6	L'héritage	98
9.1.7	Quelques remarques au sujet de @ISA	98
9.1.8	Surcharge de méthodes	99
9.1.9	Effectuer la recherche à partir d'un point différent	99
9.1.10	Le SUPER moyen de faire des choses	100
9.1.11	Où en sommes-nous ?	100
9.1.12	Un cheval est un cheval bien sûr... Mais n'est-il que cela ?	101
9.1.13	Appel d'une méthode d'instance	101
9.1.14	Accès aux données d'instance	102
9.1.15	Comment fabriquer un cheval	102
9.1.16	Héritage de constructeur	103
9.1.17	Concevoir une méthode qui marche aussi bien avec des instances qu'avec des classes	103
9.1.18	Ajout de paramètres aux méthodes	104
9.1.19	Des instances plus intéressantes	105
9.1.20	Des chevaux de couleurs différentes	105
9.1.21	Résumé	106
9.2	VOIR AUSSI	106
9.3	COPYRIGHT	106
9.4	TRADUCTION	106
9.4.1	Version	106
9.4.2	Traducteur	106
9.4.3	Relecture	106
<b>10</b>	<b>perltoot</b>	<b>107</b>
10.1	DESCRIPTION	107
10.2	Créer une classe	107
10.2.1	Représentation des objets	108
10.2.2	L'interface d'une classe	108
10.2.3	Constructeurs et méthodes d'objet	109
10.2.4	En prévision du futur: de meilleurs constructeurs	110
10.2.5	Destructeurs	110
10.2.6	Autres méthodes d'objets	111
10.3	Données de classe	111
10.3.1	Accès aux données de classe	112
10.3.2	Méthodes de débogage	113
10.3.3	Destructeurs de classes	114
10.3.4	La documentation de l'interface	114
10.4	Agrégation	115
10.5	Héritage	117
10.5.1	Polymorphisme	119

10.5.2 Héritage multiple . . . . .	121
10.5.3 UNIVERSAL: la racine de tous les objets . . . . .	122
10.5.4 Approfondissements de quelques détails de UNIVERSAL . . . . .	123
10.6 Autre représentation d'objet . . . . .	123
10.6.1 Des objets sous forme de tableaux . . . . .	124
10.6.2 Des objets sous forme de fermeture (closure) . . . . .	125
10.7 AUTOLOAD: les méthodes mandataires . . . . .	126
10.7.1 Méthodes auto-chargées d'accès aux données . . . . .	126
10.7.2 Méthodes d'accès aux données auto-chargées et héritées . . . . .	127
10.8 Méta-outils classiques . . . . .	128
10.8.1 Class::Struct . . . . .	128
10.8.2 Les données membres comme des variables . . . . .	130
10.9 NOTES . . . . .	131
10.9.1 Terminologie objet . . . . .	131
10.10 VOIR AUSSI . . . . .	132
10.11 AUTEURS ET COPYRIGHT . . . . .	132
10.11.1 Remerciements . . . . .	132
10.12 TRADUCTION . . . . .	132
10.12.1 Version . . . . .	132
10.12.2 Traducteur . . . . .	132
10.12.3 Relecture . . . . .	132
<b>11 perltooc</b> . . . . .	<b>133</b>
11.1 DESCRIPTION . . . . .	133
11.2 Données de classe prêtes à l'emploi . . . . .	134
11.3 Données de classe en tant que variables de paquetage . . . . .	134
11.3.1 Mettre tous ses oeufs dans le même panier . . . . .	135
11.3.2 À propos de l'héritage . . . . .	135
11.3.3 Le méta-objet éponyme . . . . .	136
11.3.4 Références indirectes aux données de classe . . . . .	138
11.3.5 Les classes univalentes . . . . .	140
11.3.6 Les attributs transparents . . . . .	142
11.4 Données de classe en tant que variables lexicales . . . . .	145
11.4.1 Domaine privé et responsabilité . . . . .	145
11.4.2 Variables lexicales dans la portée du fichier . . . . .	145
11.4.3 Retour sur l'héritage . . . . .	147
11.4.4 Fermer la porte et jeter la clé . . . . .	148
11.4.5 Retour sur la transparence . . . . .	149
11.5 NOTES . . . . .	151
11.6 VOIR AUSSI . . . . .	151
11.7 AUTEUR ET COPYRIGHT . . . . .	151
11.8 REMERCIEMENTS . . . . .	151
11.9 HISTORIQUE . . . . .	151
11.10 TRADUCTION . . . . .	151
11.10.1 Version . . . . .	151
11.10.2 Traducteur . . . . .	151
11.10.3 Relecture . . . . .	151



<b>12 perlbot</b>	<b>152</b>
12.1 DESCRIPTION	152
12.2 SÉRIE DE CONSEILS OO	152
12.3 VARIABLES D'INSTANCE	153
12.4 VARIABLES D'INSTANCE SCALAIRES	153
12.5 HÉRITAGE DE VARIABLES D'INSTANCE	154
12.6 RELATIONS ENTRE OBJETS	154
12.7 REMPLACER DES MÉTHODES DE CLASSES DE BASE	155
12.8 UTILISATION DE RELATIONS SDBM	155
12.9 PRÉVOIR LA RÉUTILISATION DU CODE	156
12.10OBJET ET CONTEXTE DE CLASSE	158
12.11HÉRITAGE D'UN CONSTRUCTEUR	159
12.12DÉLÉGATION	159
12.13VOIR AUSSI	160
12.14TRADUCTION	160
12.14.1 Version	160
12.14.2 Traducteurs	160
12.14.3 Relecture	160
<b>13 perlstyle</b>	<b>161</b>
13.1 DESCRIPTION	161
13.2 TRADUCTION	163
13.2.1 Version	163
13.2.2 Traducteur	163
13.2.3 Relecture	163
<b>14 perlcheat</b>	<b>164</b>
14.1 DESCRIPTION	164
14.1.1 L'anti-sèche	164
14.2 REMERCIEMENTS	165
14.3 AUTEUR	165
14.4 VOIR AUSSI	165
14.5 TRADUCTION	165
14.5.1 Version	165
14.5.2 Traducteur	165
14.5.3 Relecture	166
<b>15 perltrap</b>	<b>167</b>
15.1 DESCRIPTION	167
15.1.1 Pièges de awk	167
15.1.2 Pièges de C	168
15.1.3 Pièges de sed	168
15.1.4 Pièges du shell	168
15.1.5 Pièges de Perl	169
15.1.6 Pièges entre Perl4 et Perl5	169
15.1.7 Pièges liés aux corrections de bugs, aux désapprobations et aux abandons	170
15.1.8 Pièges de l'analyse syntaxique	172
15.1.9 Pièges numériques	173

15.1.10 Pièges des types de données généraux . . . . .	173
15.1.11 Pièges du contexte - contextes scalaires et de liste . . . . .	175
15.1.12 Pièges de la précédence (les priorités) . . . . .	176
15.1.13 Pièges des expressions rationnelles générales lors de l'utilisation de s///, etc. . . . .	177
15.1.14 Pièges des sous-programmes, des signaux et des tris . . . . .	179
15.1.15 Pièges du système d'exploitation . . . . .	179
15.1.16 Pièges de l'interpolation . . . . .	180
15.1.17 Pièges DBM . . . . .	181
15.1.18 Pièges non classés . . . . .	182
15.2 TRADUCTION . . . . .	182
15.2.1 Version . . . . .	182
15.2.2 Traducteur . . . . .	182
15.2.3 Relecture . . . . .	182
<b>16 perldebut</b> . . . . .	<b>183</b>
16.1 DESCRIPTION . . . . .	183
16.2 Utiliser strict (use strict) . . . . .	183
16.3 Observer les variables et -w et v . . . . .	184
16.4 Aide (help) . . . . .	185
16.5 Pas à pas dans le code . . . . .	189
16.6 Emplacement réservé pour a, w, t, T . . . . .	192
16.7 Expressions rationnelles (ou régulières) . . . . .	192
16.8 Conseils sur les sorties . . . . .	193
16.9 CGI . . . . .	193
16.10 Interfaces graphiques (GUI) . . . . .	193
16.11 Résumé . . . . .	193
16.12 VOIR AUSSI . . . . .	193
16.13 AUTEUR . . . . .	194
16.14 CONTRIBUTIONS . . . . .	194
16.15 TRADUCTION . . . . .	194
16.15.1 Version . . . . .	194
16.15.2 Traducteur . . . . .	194
16.15.3 Relecture . . . . .	194
<b>17 perlfaq</b> . . . . .	<b>195</b>
17.1 DESCRIPTION . . . . .	195
17.1.1 Où se procurer ce document ? . . . . .	195
17.1.2 Comment contribuer à ce document. . . . .	195
17.1.3 Que va-t-il se passer si vous envoyez votre problème de Perl aux auteurs ? . . . . .	195
17.2 CREDITS . . . . .	195
17.3 Author and Copyright Information . . . . .	196
17.4 Table des matières . . . . .	196
17.5 Toutes les questions . . . . .	196
17.6 TRADUCTION . . . . .	202
17.6.1 Version . . . . .	202
17.6.2 Traducteur . . . . .	202
17.6.3 Relecture . . . . .	202

<b>18 perlfaq1</b>	<b>203</b>
18.1 DESCRIPTION	203
18.1.1 Qu'est ce que Perl ?	203
18.1.2 Qui supporte Perl ? Qui le développe ? Pourquoi est-il gratuit ?	203
18.1.3 Quelle version de Perl dois-je utiliser ?	204
18.1.4 Qu'est-ce que veut dire perl4, perl5 ou perl6 ?	204
18.1.5 Qu'est-ce que Ponie ?	204
18.1.6 Qu'est-ce que perl6 ?	205
18.1.7 Est-ce que Perl est stable ?	205
18.1.8 Est-il difficile d'apprendre Perl ?	205
18.1.9 Est-ce que Perl tient la comparaison avec d'autres langages comme Java, Python, REXX, Scheme ou Tcl ?	205
18.1.10 Que puis-je faire avec Perl ?	206
18.1.11 Quand ne devrais-je pas programmer en Perl ?	206
18.1.12 Quelle est la différence entre "perl" et "Perl" ?	206
18.1.13 Parle-t-on de programme Perl ou de script Perl ?	206
18.1.14 Qu'est ce qu'un JAPH ?	207
18.1.15 Où peut on trouver une liste des mots d'esprit de Larry Wall ?	207
18.1.16 Comment convaincre mon administrateur système/chef de projet/employés d'utiliser Perl/Perl5 plutôt qu'un autre langage ?	207
18.2 AUTEUR ET COPYRIGHT	207
18.3 TRADUCTION	208
18.3.1 Version	208
18.3.2 Traducteur	208
18.3.3 Relecture	208
<b>19 perlfaq2</b>	<b>209</b>
19.1 DESCRIPTION	209
19.1.1 Quelles machines supportent perl ? Où puis-je trouver Perl ?	209
19.1.2 Comment trouver une version binaire de perl ?	209
19.1.3 Je n'ai pas de compilateur C sur mon système. Comment puis-je compiler mon propre interpréteur Perl ?	210
19.1.4 J'ai copié le binaire perl d'une machine sur une autre mais les scripts ne fonctionnent pas.	210
19.1.5 J'ai récupéré les sources et j'essaie de les compiler mais gdbm/dynamic loading/malloc/linking/... échoue. Comment faire pour que ça marche ?	210
19.1.6 Quels modules et extensions existent pour Perl ? Qu'est-ce que CPAN ? Que signifie CPAN/src/... ?	210
19.1.7 Existe-t-il une version de Perl certifiée ISO ou ANSI ?	211
19.1.8 Où puis-je trouver des informations sur Perl ?	211
19.1.9 Quels sont les groupes de discussion concernant Perl sur Usenet ? Où puis-je poser mes questions ?	211
19.1.10 Où puis-je poster mon code source ?	212
19.1.11 Les livres sur Perl	212
19.1.12 Quels sont les revues ou magazines parlant de Perl	215
19.1.13 Quelles sont les listes de diffusion concernant Perl ?	215
19.1.14 Où trouver les archives de comp.lang.perl.misc ?	215
19.1.15 Où puis-je acheter une version commerciale de Perl ?	215
19.1.16 Où dois-je envoyer mes rapports de bugs ?	215
19.1.17 Qu'est-ce que perl.com ? Les Perl Mongers ? pm.org ? perl.org ? cpan.org ?	216
19.2 TRADUCTION	216
19.2.1 Version	216
19.2.2 Traducteur	216
19.2.3 Relecture	216

<b>20 perlfaq3</b>	<b>217</b>
20.1 DESCRIPTION	217
20.1.1 Comment fais-je pour... ?	217
20.1.2 Comment utiliser Perl de façon interactive ?	217
20.1.3 Existe-t-il un shell Perl ?	217
20.1.4 Comment puis-je connaître les modules installés sur mon système ?	218
20.1.5 Comment déboguer mes programmes Perl ?	218
20.1.6 Comment mesurer les performances de mes programmes Perl ?	219
20.1.7 Comment faire une liste croisée des appels de mon programme Perl ?	219
20.1.8 Existe-t-il un outil de mise en page de code Perl ?	219
20.1.9 Existe-t-il un ctags pour Perl ?	220
20.1.10 Existe-t-il un environnement de développement intégré (IDE) ou un éditeur Perl sous Windows ?	220
20.1.11 Où puis-je trouver des macros pour Perl sous vi ?	222
20.1.12 Où puis-je trouver le mode perl pour emacs ?	222
20.1.13 Comment utiliser des 'curses' avec Perl ?	222
20.1.14 Comment puis-je utiliser X ou Tk avec Perl ?	222
20.1.15 Comment rendre mes programmes Perl plus rapides ?	223
20.1.16 Comment faire pour que mes programmes Perl occupent moins de mémoire ?	223
20.1.17 Est-ce sûr de retourner un pointeur sur une donnée locale ?	224
20.1.18 Comment puis-je libérer un tableau ou une table de hachage pour réduire mon programme ?	224
20.1.19 Comment rendre mes scripts CGI plus efficaces ?	225
20.1.20 Comment dissimuler le code source de mon programme Perl ?	225
20.1.21 Comment compiler mon programme Perl en code binaire ou C ?	226
20.1.22 Comment compiler Perl pour en faire du Java ?	226
20.1.23 Comment faire fonctionner #!perl sur [MS-DOS,NT,...] ?	226
20.1.24 Puis-je écrire des programmes Perl pratiques sur la ligne de commandes ?	226
20.1.25 Pourquoi les commandes Perl à une ligne ne fonctionnent-elles pas sur mon DOS/Mac/VMS ?	227
20.1.26 Où puis-je en apprendre plus sur la programmation CGI et Web en Perl ?	228
20.1.27 Où puis-je apprendre la programmation orientée objet en Perl ?	228
20.1.28 Où puis-je en apprendre plus sur l'utilisation liée de Perl et de C ?	228
20.1.29 J'ai lu perlembed, perlguts, etc., mais je ne peux inclure du perl dans mon programme C, qu'est ce qui ne va pas ?	228
20.1.30 Quand j'ai tenté d'exécuter mes scripts, j'ai eu ce message. Qu'est ce que cela signifie ?	228
20.1.31 Qu'est-ce que MakeMaker ?	228
20.2 AUTEUR ET COPYRIGHT	229
20.3 TRADUCTION	229
20.3.1 Version	229
20.3.2 Traducteur	229
20.3.3 Relecture	229

<b>21 perlfaq4</b>	<b>230</b>
21.1 DESCRIPTION	230
21.2 Données : nombres	230
21.2.1 Pourquoi est-ce que j'obtiens des longs nombres décimaux (ex. 19.9499999999999) à la place du nombre que j'attends (ex. 19.95) ?	230
21.2.2 Pourquoi int() ne fonctionne pas bien ?	230
21.2.3 Pourquoi mon nombre octal n'est-il pas interprété correctement ?	231
21.2.4 Perl a-t-il une fonction round() ? Et ceil() (majoration) et floor() (minoration) ? Et des fonctions trigonométriques ?	231
21.2.5 Comment faire des conversions numériques entre différentes bases, entre différentes représentations ?	232
21.2.6 Pourquoi & ne fonctionne-t-il pas comme je le veux ?	233
21.2.7 Comment multiplier des matrices ?	234
21.2.8 Comment effectuer une opération sur une série d'entiers ?	234
21.2.9 Comment produire des chiffres romains ?	234
21.2.10 Pourquoi mes nombres aléatoires ne sont-ils pas aléatoires ?	234
21.2.11 Comment obtenir un nombre aléatoire entre X et Y ?	235
21.3 Données : dates	235
21.3.1 Comment trouver le jour ou la semaine de l'année ?	235
21.3.2 Comment trouver le siècle ou le millénaire actuel ?	235
21.3.3 Comment comparer deux dates ou en calculer la différence ?	236
21.3.4 Comment convertir une chaîne de caractères en secondes depuis l'origine des temps ?	236
21.3.5 Comment trouver le jour du calendrier Julien ?	236
21.3.6 Comment trouver la date d'hier ?	236
21.3.7 Perl a-t-il un problème avec l'an 2000 ? Perl est-il compatible an 2000 ?	237
21.4 Données : chaînes de caractères	237
21.4.1 Comment m'assurer de la validité d'une entrée ?	237
21.4.2 Comment supprimer les caractères d'échappement d'une chaîne de caractères ?	237
21.4.3 Comment enlever des paires de caractères successifs ?	237
21.4.4 Comment effectuer des appels de fonction dans une chaîne ?	238
21.4.5 Comment repérer des éléments appariés ou imbriqués ?	238
21.4.6 Comment inverser une chaîne de caractères ?	239
21.4.7 Comment développer les tabulations dans une chaîne de caractères ?	239
21.4.8 Comment remettre en forme un paragraphe ?	239
21.4.9 Comment accéder à ou modifier N caractères d'une chaîne de caractères ?	239
21.4.10 Comment changer la nième occurrence de quelque chose ?	240
21.4.11 Comment compter le nombre d'occurrences d'une sous-chaîne dans une chaîne de caractères ?	240
21.4.12 Comment mettre en majuscule toutes les premières lettre des mots d'une ligne ?	241
21.4.13 Comment découper une chaîne séparée par un [caractère] sauf à l'intérieur d'un [caractère] ?	241
21.4.14 Comment supprimer des espaces blancs au début/à la fin d'une chaîne ?	242
21.4.15 Comment cadrer une chaîne avec des blancs ou un nombre avec des zéros ?	243
21.4.16 Comment extraire une sélection de colonnes d'une chaîne de caractères ?	243
21.4.17 Comment calculer la valeur soundex d'une chaîne ?	244
21.4.18 Comment interpoler des variables dans des chaînes de texte ?	244
21.4.19 En quoi est-ce un problème de toujours placer "\$vars" entre guillemets ?	244
21.4.20 Pourquoi est-ce que mes documents <<HERE ne marchent pas ?	245
21.5 Données : tableaux	246
21.5.1 Quelle est la différence entre une liste et un tableau ?	246

21.5.2	Quelle est la différence entre \$array[1] et @array[1] ?	246
21.5.3	Comment supprimer les doublons d'une liste ou d'un tableau ?	246
21.5.4	Comment savoir si une liste ou un tableau inclut un certain élément ?	247
21.5.5	Comment calculer la différence entre deux tableaux ? Comment calculer l'intersection entre deux tableaux ?	248
21.5.6	Comment tester si deux tableaux ou hachages sont égaux ?	248
21.5.7	Comment trouver le premier élément d'un tableau vérifiant une condition donnée ?	249
21.5.8	Comment gérer des listes chaînées ?	250
21.5.9	Comment gérer des listes circulaires ?	250
21.5.10	Comment mélanger le contenu d'un tableau ?	251
21.5.11	Comment traiter/modifier chaque élément d'un tableau ?	251
21.5.12	Comment sélectionner aléatoirement un élément d'un tableau ?	252
21.5.13	Comment générer toutes les permutations des N éléments d'une liste ?	252
21.5.14	Comment trier un tableau par (n'importe quoi) ?	253
21.5.15	Comment manipuler des tableaux de bits ?	253
21.5.16	Pourquoi defined() retourne vrai sur des tableaux et hachages vides ?	255
21.6	Données : tables de hachage (tableaux associatifs)	255
21.6.1	Comment traiter une table entière ?	255
21.6.2	Que se passe-t-il si j'ajoute ou j'enlève des clefs d'un hachage pendant que j'y itère dessus ?	255
21.6.3	Comment rechercher un élément d'un hachage par sa valeur ?	255
21.6.4	Comment savoir combien d'entrées sont dans un hachage ?	256
21.6.5	Comment trier une table de hachage (par valeur ou par clef) ?	256
21.6.6	Comment conserver mes tables de hachage dans l'ordre ?	257
21.6.7	Quelle est la différence entre "delete" et "undef" pour des tables de hachage ?	257
21.6.8	Pourquoi mes tables de hachage liées (par tie()) ne font pas la distinction entre exists et defined ?	258
21.6.9	Comment réinitialiser une opération each() non terminée ?	258
21.6.10	Comment obtenir l'unicité des clefs de deux hachages ?	258
21.6.11	Comment enregistrer un tableau multidimensionnel dans un fichier DBM ?	259
21.6.12	Comment faire en sorte que mon hachage conserve l'ordre des éléments que j'y mets ?	259
21.6.13	Pourquoi le passage à un sous-programme d'un élément non défini d'un hachage le crée du même coup ?	259
21.6.14	Comment faire l'équivalent en Perl d'une structure en C, d'une classe/d'un hachage en C++ ou d'un tableau de hachages ou de tableaux ?	259
21.6.15	Comment utiliser une référence comme clef d'une table de hachage ?	259
21.7	Données : divers	260
21.7.1	Comment manipuler proprement des données binaires ?	260
21.7.2	Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ?	260
21.7.3	Comment conserver des données persistantes entre divers appels de programme ?	261
21.7.4	Comment afficher ou copier une structure de données récursive ?	261
21.7.5	Comment définir des méthodes pour toutes les classes ou tous les objets ?	261
21.7.6	Comment vérifier la somme de contrôle d'une carte de crédit ?	261
21.7.7	Comment compacter des tableaux de nombres à virgule flottante simples ou doubles pour le code XS ?	261
21.8	AUTEUR ET COPYRIGHT	261
21.9	TRADUCTION	261
21.9.1	Version	262
21.9.2	Traducteur	262
21.9.3	Relecture	262

<b>22 perlfaq5</b>	<b>263</b>
22.1 DESCRIPTION	263
22.1.1 Comment vider ou désactiver les tampons en sortie ? Pourquoi m'en soucier ?	263
22.1.2 Comment changer une ligne, effacer une ligne, insérer une ligne au milieu ou ajouter une ligne en tête d'un fichier ?	264
22.1.3 Comment déterminer le nombre de lignes d'un fichier ?	264
22.1.4 Comment utiliser l'option <code>-i</code> de Perl depuis l'intérieur d'un programme ?	264
22.1.5 Comment puis-je copier un fichier ?	264
22.1.6 Comment créer un fichier temporaire ?	265
22.1.7 Comment manipuler un fichier avec des enregistrements de longueur fixe ?	265
22.1.8 Comment rendre un descripteur de fichier local à une routine ? Comment passer des descripteurs à d'autres routines ? Comment construire un tableau de descripteurs ?	266
22.1.9 Comment utiliser un descripteur de fichier indirectement ?	266
22.1.10 Comment mettre en place un pied-de-page avec <code>write()</code> ?	268
22.1.11 Comment rediriger un <code>write()</code> dans une chaîne ?	268
22.1.12 Comment afficher mes nombres avec des virgules pour délimiter les milliers ?	268
22.1.13 Comment traduire les tildes ( <code>~</code> ) dans un nom de fichier ?	268
22.1.14 Pourquoi les fichiers que j'ouvre en lecture-écriture se voient-ils effacés ?	269
22.1.15 Pourquoi <code>&lt;*&gt;</code> donne de temps en temps l'erreur "Argument list too long" ?	270
22.1.16 Y a-t-il une fuite / un bug avec <code>glob()</code> ?	270
22.1.17 Comment ouvrir un fichier dont le nom commence par <code>&gt;</code> ou se termine par des espaces ?	270
22.1.18 Comment renommer un fichier de façon sûre ?	270
22.1.19 Comment verrouiller un fichier ?	270
22.1.20 Pourquoi ne pas faire simplement <code>open(FH, "&gt;file.lock")</code> ?	271
22.1.21 Je ne comprends toujours pas le verrouillage. Je veux seulement incrémenter un compteur dans un fichier. Comment faire ?	271
22.1.22 Je souhaite juste ajouter un peu de texte à la fin d'un fichier. Dois-je tout de même utiliser le verrouillage ?	272
22.1.23 Comment modifier un fichier binaire directement ?	272
22.1.24 Comment récupérer la date d'un fichier en perl ?	272
22.1.25 Comment modifier la date d'un fichier en perl ?	273
22.1.26 Comment écrire dans plusieurs fichiers simultanément ?	273
22.1.27 Comment lire le contenu d'un fichier d'un seul coup ?	273
22.1.28 Comment lire un fichier paragraphe par paragraphe ?	274
22.1.29 Comment lire un seul caractère d'un fichier ? Et du clavier ?	274
22.1.30 Comment savoir si un caractère est disponible sur un descripteur de fichier ?	275
22.1.31 Comment écrire un <code>tail -f</code> en perl ?	276
22.1.32 Comment faire un <code>dup()</code> sur un descripteur en Perl ?	277
22.1.33 Comment fermer un descripteur connu par son numéro ?	277
22.1.34 Pourquoi <code>"C:\temp\foo"</code> n'indique pas un fichier DOS ? Et même <code>"C:\temp\foo.exe"</code> ne marche pas ?	277
22.1.35 Pourquoi <code>glob("*.*)"</code> ne donne-t-il pas tous les fichiers ?	277
22.1.36 Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi <code>-i</code> écrit-il dans des fichiers protégés ? N'est-ce pas un bug de Perl ?	278
22.1.37 Comment sélectionner une ligne au hasard dans un fichier ?	278
22.1.38 Pourquoi obtient-on des espaces étranges lorsqu'on affiche un tableau de lignes ?	278
22.2 AUTHOR AND COPYRIGHT (on Original English Version)	278
22.3 TRADUCTION	279
22.3.1 Version	279
22.3.2 Traducteur	279
22.3.3 Relecture	279

<b>23 perlfaq6</b>	<b>280</b>
23.1 DESCRIPTION	280
23.1.1 Comment utiliser les expressions rationnelles sans créer du code illisible et difficile à maintenir ?	280
23.1.2 J'ai des problèmes pour faire une reconnaissance sur plusieurs lignes. Qu'est-ce qui ne va pas ?	281
23.1.3 Comment extraire des lignes entre deux motifs qui sont chacun sur des lignes différentes ?	282
23.1.4 J'ai mis une expression rationnelle dans \$/ mais cela ne marche pas. Qu'est-ce qui est faux ?	282
23.1.5 Comment faire une substitution indépendante de la casse de la partie gauche mais préservant cette casse dans la partie droite ?	283
23.1.6 Comment faire pour que \w reconnaisse les caractères nationaux ?	284
23.1.7 Comment reconnaître une version locale-équivalente de /[a-zA-Z]/ ?	284
23.1.8 Comment protéger une variable pour l'utiliser dans une expression rationnelle ?	284
23.1.9 À quoi sert vraiment /o ?	285
23.1.10 Comment utiliser une expression rationnelle pour enlever les commentaires de type C d'un fichier ?	285
23.1.11 Est-ce possible d'utiliser les expressions rationnelles de Perl pour reconnaître du texte bien équilibré ?	286
23.1.12 Que veut dire "les expressions rationnelles sont gourmandes" ? Comment puis-je le contourner ?	286
23.1.13 Comment examiner chaque mot dans chaque ligne ?	287
23.1.14 Comment afficher un rapport sur les fréquences de mots ou de lignes ?	287
23.1.15 Comment faire une reconnaissance approximative ?	287
23.1.16 Comment reconnaître efficacement plusieurs expressions rationnelles en même temps ?	287
23.1.17 Pourquoi les recherches de limite de mot avec \b ne marchent pas pour moi ?	288
23.1.18 Pourquoi l'utilisation de \$&, \$', or \$' ralentit tant mon programme ?	289
23.1.19 À quoi peut bien servir \G dans une expression rationnelle ?	289
23.1.20 Les expressions rationnelles de Perl sont-elles AFD ou AFN ? Sont-elles conformes à POSIX ?	291
23.1.21 Qu'est ce qui ne va pas avec l'utilisation de grep dans un contexte vide ?	291
23.1.22 Comment reconnaître des chaînes avec des caractères multi-octets ?	291
23.1.23 Comment rechercher un motif fourni par l'utilisateur ?	292
23.2 AUTEUR ET COPYRIGHT	292
23.3 TRADUCTION	292
23.3.1 Version	292
23.3.2 Traducteur	292
23.3.3 Relecture	292
<b>24 perlfaq7</b>	<b>293</b>
24.1 DESCRIPTION	293
24.1.1 Puis-je avoir une BNF/yacc/RE pour le langage Perl ?	293
24.1.2 Quels sont tous ces \$@%&* de signes de ponctuation, et comment savoir quand les utiliser ?	293
24.1.3 Dois-je toujours/jamais mettre mes chaînes entre guillemets ou utiliser les points-virgules et les virgules ?	294
24.1.4 Comment ignorer certaines valeurs de retour ?	294
24.1.5 Comment bloquer temporairement les avertissements ?	294
24.1.6 Qu'est-ce qu'une extension ?	295
24.1.7 Pourquoi les opérateurs de Perl ont-ils une précedence différente de celle des opérateurs en C ?	295
24.1.8 Comment déclarer/créer une structure ?	295
24.1.9 Comment créer un module ?	296
24.1.10 Comment créer une classe ?	296
24.1.11 Comment déterminer si une variable est souillée ?	296
24.1.12 Qu'est-ce qu'une fermeture ?	296



24.1.13	Qu'est-ce que le suicide de variable et comment le prévenir ?	297
24.1.14	Comment passer/renvoyer {une fonction, un handle de fichier, un tableau, un hachage, une méthode, une expression rationnelle} ?	297
24.1.15	Comment créer une variable statique ?	299
24.1.16	Quelle est la différence entre la portée dynamique et lexicale (statique) ? Entre local() et my() ?	300
24.1.17	Comment puis-je accéder à une variable dynamique lorsqu'un lexical de même nom est à portée ?	300
24.1.18	Quelle est la différence entre les liaisons profondes et superficielles ?	301
24.1.19	Pourquoi "my(\$foo) = <FICHIER>;" ne marche pas ?	301
24.1.20	Comment redéfinir une fonction, un opérateur ou une méthode prédéfini ?	301
24.1.21	Quelle est la différence entre l'appel d'une fonction par &foo et par foo() ?	301
24.1.22	Comment créer une instruction switch ou case ?	302
24.1.23	Comment intercepter les accès aux variables, aux fonctions, aux méthodes indéfinies ?	303
24.1.24	Pourquoi une méthode incluse dans ce même fichier ne peut-elle pas être trouvée ?	303
24.1.25	Comment déterminer mon paquetage courant ?	303
24.1.26	Comment commenter un grand bloc de code perl ?	304
24.1.27	Comment supprimer un paquetage ?	304
24.1.28	Comment utiliser une variable comme nom de variable ?	304
24.1.29	Que signifie "bad interpreter" ?	306
24.2	AUTEUR ET COPYRIGHT	306
24.3	TRADUCTION	306
24.3.1	Version	306
24.3.2	Traducteur	306
24.3.3	Relecture	306
<b>25</b>	<b>perlfaq8</b>	<b>307</b>
25.1	DESCRIPTION	307
25.1.1	Comment savoir sur quel système d'exploitation je tourne ?	307
25.1.2	Pourquoi ne revient-on pas après un exec() ?	307
25.1.3	Comment utiliser le clavier/écran/souris de façon élaborée ?	307
25.1.4	Comment afficher quelque chose en couleur ?	308
25.1.5	Comment lire simplement une touche sans attendre un appui sur "entrée" ?	308
25.1.6	Comment vérifier si des données sont en attente depuis le clavier ?	309
25.1.7	Comment effacer l'écran ?	309
25.1.8	Comment obtenir la taille de l'écran ?	310
25.1.9	Comment demander un mot de passe à un utilisateur ?	310
25.1.10	Comment lire et écrire sur le port série ?	310
25.1.11	Comment décoder les fichiers de mots de passe cryptés ?	311
25.1.12	Comment lancer un processus en arrière plan ?	312
25.1.13	Comment capturer un caractère de contrôle, un signal ?	312
25.1.14	Comment modifier le fichier masqué (shadow) de mots de passe sous Unix ?	313
25.1.15	Comment positionner l'heure et la date ?	313
25.1.16	Comment effectuer un sleep() ou alarm() de moins d'une seconde ?	313
25.1.17	Comment mesurer un temps inférieur à une seconde ?	313
25.1.18	Comment réaliser un atexit() ou setjmp()/longjmp() ? (traitement d'exceptions)	314
25.1.19	Pourquoi mes programmes avec socket() ne marchent pas sous System V (Solaris) ? Que signifie le message d'erreur « Protocole non supporté » ?	314
25.1.20	Comment appeler les fonctions C spécifiques à mon système depuis Perl ?	314
25.1.21	Où trouver les fichiers d'inclusion pour ioctl() et syscall() ?	314

25.1.22 Pourquoi les scripts perl en setuid se plaignent-ils d'un problème noyau ?	315
25.1.23 Comment ouvrir un tube depuis et vers une commande simultanément ?	315
25.1.24 Pourquoi ne puis-je pas obtenir la sortie d'une commande avec system() ?	315
25.1.25 Comment capturer la sortie STDERR d'une commande externe ?	315
25.1.26 Pourquoi open() ne retourne-t-il pas d'erreur lorsque l'ouverture du tube échoue ?	317
25.1.27 L'utilisation des apostrophes inversées dans un contexte vide pose-t-elle problème ?	318
25.1.28 Comment utiliser des apostrophes inversées sans traitement du shell ?	318
25.1.29 Pourquoi mon script ne lit-il plus rien de STDIN après que je lui ai envoyé EOF (^D sur Unix, ^Z sur MS-DOS) ?	319
25.1.30 Comment convertir mon script shell en perl ?	319
25.1.31 Puis-je utiliser perl pour lancer une session telnet ou ftp ?	319
25.1.32 Comment écrire "expect" en Perl ?	319
25.1.33 Peut-on cacher les arguments de perl sur la ligne de commande aux programmes comme "ps" ?	319
25.1.34 J'ai {changé de répertoire, modifié mon environnement} dans un script perl. Pourquoi les changements disparaissent-ils lorsque le script se termine ? Comment rendre mes changements visibles ?	320
25.1.35 Comment fermer le descripteur de fichier attaché à un processus sans attendre que ce dernier se termine ?	320
25.1.36 Comment lancer un processus démon ?	320
25.1.37 Comment savoir si je tourne de façon interactive ou pas ?	320
25.1.38 Comment sortir d'un blocage sur événement lent ?	320
25.1.39 Comment limiter le temps CPU ?	321
25.1.40 Comment éviter les processus zombies sur un système Unix ?	321
25.1.41 Comment utiliser une base de données SQL ?	321
25.1.42 Comment terminer un appel à system() avec un control-C ?	321
25.1.43 Comment ouvrir un fichier sans bloquer ?	321
25.1.44 Comment faire la différence entre les erreurs shell et les erreurs perl ?	321
25.1.45 Comment installer un module du CPAN ?	322
25.1.46 Quelle est la différence entre require et use ?	323
25.1.47 Comment gérer mon propre répertoire de modules/bibliothèques ?	323
25.1.48 Comment ajouter le répertoire dans lequel se trouve mon programme dans le chemin de recherche des modules / bibliothèques ?	323
25.1.49 Comment ajouter un répertoire dans mon chemin de recherche (@INC) à l'exécution ?	323
25.1.50 Qu'est-ce que socket.ph et où l'obtenir ?	324
25.2 AUTEURS	324
25.3 TRADUCTION	324
25.3.1 Version	324
25.3.2 Traducteur	324
25.3.3 Relecture	324

<b>26 perlfaq9</b>	<b>325</b>
26.1 DESCRIPTION	325
26.1.1 Quelle est la forme correcte d'une réponse d'un script CGI ?	325
26.1.2 Mon script CGI fonctionne en ligne de commandes mais pas depuis un navigateur. (500 Server Error)	325
26.1.3 Comment faire pour obtenir de meilleurs messages d'erreur d'un programme CGI ?	326
26.1.4 Comment enlever les balises HTML d'une chaîne ?	326
26.1.5 Comment extraire des URL ?	327
26.1.6 Comment télécharger un fichier depuis la machine d'un utilisateur ? Comment ouvrir un fichier d'une autre machine ?	327
26.1.7 Comment faire un menu pop-up en HTML ?	327
26.1.8 Comment récupérer un fichier HTML ?	328
26.1.9 Comment automatiser la soumission d'un formulaire HTML ?	328
26.1.10 Comment décoder ou créer ces %-encodings sur le web ?	329
26.1.11 Comment rediriger le navigateur vers une autre page ?	329
26.1.12 Comment mettre un mot de passe sur mes pages Web ?	329
26.1.13 Comment éditer mes fichiers .htpasswd et .htgroup en Perl ?	329
26.1.14 Comment être sûr que les utilisateurs ne peuvent pas entrer de valeurs dans un formulaire qui font faire de vilaines choses à mon script CGI ?	330
26.1.15 Comment analyser un en-tête de mail ?	330
26.1.16 Comment décoder un formulaire CGI ?	330
26.1.17 Comment vérifier la validité d'une adresse électronique ?	331
26.1.18 Comment décoder une chaîne MIME/BASE64 ?	331
26.1.19 Comment renvoyer l'adresse électronique de l'utilisateur ?	332
26.1.20 Comment envoyer un mail ?	332
26.1.21 Comment utiliser MIME pour attacher des documents à un mail ?	333
26.1.22 Comment lire du courrier ?	333
26.1.23 Comment trouver mon nom de machine / nom de domaine / mon adresse IP ?	334
26.1.24 Comment récupérer un article de news ou les groupes actifs ?	334
26.1.25 Comment récupérer/envoyer un fichier par FTP ?	334
26.1.26 Comment faire du RPC en Perl ?	334
26.2 AUTEUR ET COPYRIGHT	334
26.3 TRADUCTION	334
26.3.1 Version	334
26.3.2 Traducteur	335
26.3.3 Relecture	335
<b>III Manuel de référence</b>	<b>336</b>
<b>27 perlsyn</b>	<b>337</b>
27.1 DESCRIPTION	337
27.1.1 Déclarations	337
27.1.2 Commentaires	338
27.1.3 Instructions simples	338
27.1.4 Le vrai et le faux	338
27.1.5 Modificateurs d'instruction	338
27.1.6 Instructions composées	339
27.1.7 Contrôle de boucle	340

27.1.8	Boucles for	341
27.1.9	Boucles foreach	342
27.1.10	BLOCs de base et instruction switch	343
27.1.11	Goto	345
27.1.12	POD : documentation intégrée	346
27.1.13	Bons Vieux Commentaires (Non !)	346
27.2	TRADUCTION	347
27.2.1	Version	347
27.2.2	Traducteur	347
27.2.3	RELECTURE	347
<b>28</b>	<b>perldata</b>	<b>348</b>
28.1	DESCRIPTION	348
28.1.1	Noms des variables	348
28.1.2	Contexte	349
28.1.3	Valeurs scalaires	350
28.1.4	Constructeurs de valeurs scalaires	351
28.1.5	Constructeurs de listes de valeurs	353
28.1.6	Indices	356
28.1.7	Tranches	356
28.1.8	Typeglobs et handles de Fichiers	358
28.2	VOIR AUSSI	359
28.3	TRADUCTION	359
28.3.1	Version	359
28.3.2	Traducteur	359
28.3.3	Relecture	359
<b>29</b>	<b>perlop</b>	<b>360</b>
29.1	DESCRIPTION	360
29.1.1	Priorité des opérateurs et associativité	360
29.1.2	Termes et opérateurs de listes (leftward)	361
29.1.3	L'opérateur flèche	361
29.1.4	Auto-incrémentation et auto-décrémentation	362
29.1.5	Puissance	362
29.1.6	Opérateurs symboliques unaires	362
29.1.7	Opérateurs d'application d'expressions rationnelles	363
29.1.8	Opérateurs type multiplication	363
29.1.9	Opérateurs type addition	363
29.1.10	Opérateurs de décalages	363
29.1.11	Opérateurs unaires nommés	364
29.1.12	Opérateurs de comparaisons	364
29.1.13	Opérateurs d'égalité	364
29.1.14	Opérateur et bit à bit	365
29.1.15	Opérateurs ou et ou exclusif bit à bit	365
29.1.16	Et logique style C	365
29.1.17	Défini-ou logique style C	365
29.1.18	Ou logique style C	365
29.1.19	Opérateurs d'intervalle	366

29.1.20	Opérateur conditionnel	368
29.1.21	Opérateurs d'affectation	368
29.1.22	Opérateur virgule	369
29.1.23	Opérateurs de listes (Rightward)	369
29.1.24	Non (not) logique	370
29.1.25	Et (and) logique	370
29.1.26	Ou (or), ou exclusif (xor) et défini-ou (err) logiques	370
29.1.27	Opérateurs C manquant en Perl	370
29.1.28	Opérateurs apostrophe et apparentés	371
29.1.29	Opérateurs d'expression rationnelle	372
29.1.30	Les détails sordides de l'interprétation des chaînes	381
29.1.31	Opérateurs d'E/S	383
29.1.32	Traitement des constantes	386
29.1.33	No-ops (opération vide)	386
29.1.34	Opérateurs bit à bit sur les chaînes	386
29.1.35	Arithmétique entière	387
29.1.36	Arithmétique en virgule flottante	387
29.1.37	Les grands nombres	388
29.2	TRADUCTION	388
29.2.1	Version	388
29.2.2	Traducteur	388
29.2.3	Relecture	388
<b>30</b>	<b>perlsub</b>	<b>389</b>
30.1	SYNOPSIS	389
30.2	DESCRIPTION	390
30.2.1	Variables privées via my()	392
30.2.2	Variables privées persistantes	394
30.2.3	Valeurs temporaires via local()	395
30.2.4	Lvalue et sous-programme	398
30.2.5	Passage d'entrées de table de symbole (typeglobs)	399
30.2.6	Quand faut-il encore utiliser local()	399
30.2.7	Passage par référence	400
30.2.8	Prototypes	402
30.2.9	Fonctions constantes	404
30.2.10	Surcharges des fonctions prédéfinies	405
30.2.11	Autochargement	407
30.2.12	Attributs de sous-programme	408
30.3	VOIR AUSSI	408
30.4	TRADUCTION	408
30.4.1	Version	408
30.4.2	Traducteur	408
30.4.3	Relecture	408

<b>31 perlfunc</b>	<b>409</b>
31.1 DESCRIPTION	409
31.1.1 Fonctions Perl par catégories	410
31.1.2 Portabilité	411
31.1.3 Fonctions Perl par ordre alphabétique	411
31.2 TRADUCTION	487
31.2.1 Version	487
31.2.2 Traducteur	487
31.2.3 Relecture	487
<b>32 perlopentut</b>	<b>488</b>
32.1 DESCRIPTION	488
32.2 Open à la shell	488
32.2.1 Opens simples	488
32.2.2 Handles de fichier indirects	490
32.2.3 L'ouverture des tubes	490
32.2.4 Le fichier moins ou tiret	491
32.2.5 Mélanger la lecture et l'écriture	491
32.2.6 Les filtres	491
32.3 Ouverture à la C	493
32.3.1 Permissions à la mode Unix	494
32.4 Trucs obscurs avec open	494
32.4.1 Ré-ouverture de fichiers (dups)	494
32.4.2 Exorciser la magie	495
32.4.3 Chemins d'accès comme opens	496
32.4.4 Open à un seul argument	496
32.4.5 Jouer avec STDIN et STDOUT	496
32.5 Autres considérations sur les E/S	497
32.5.1 Ouvrir des fichiers qui n'en sont pas	497
32.5.2 Ouvrir des tubes nommés	498
32.5.3 Ouvrir des Sockets	499
32.5.4 Fichiers binaires	499
32.5.5 Verrouillage de fichiers	499
32.5.6 Couches d'entrées/sorties	501
32.6 VOIR AUSSI	501
32.7 AUTEUR ET COPYRIGHT	501
32.8 TRADUCTION	501
32.8.1 Version	501
32.8.2 Traducteur	502
32.8.3 Relecture	502
32.9 HISTORIQUE	502

<b>33 perlpod</b>	<b>503</b>
33.1 DESCRIPTION	503
33.1.1 Les paragraphes ordinaires	503
33.1.2 Les paragraphes verbatim	503
33.1.3 Les paragraphes de commande	503
33.1.4 Codes de mise en forme	507
33.1.5 L'objectif	509
33.1.6 Incorporer du Pod dans les modules Perl	509
33.1.7 Conseils pour écrire en Pod	509
33.2 VOIR AUSSI	510
33.3 AUTEUR	510
33.4 TRADUCTION	510
33.4.1 Version	510
33.4.2 Traducteur	510
33.4.3 Relecture	510
<b>34 perlrun</b>	<b>511</b>
34.1 SYNOPSIS	511
34.2 DESCRIPTION	511
34.2.1 #! et l'isolement (quoting) sur les systèmes non-Unix	512
34.2.2 Emplacement de Perl	513
34.2.3 Options de ligne de commandes	514
34.3 ENVIRONNEMENT	521
34.4 TRADUCTION	525
34.4.1 Version	525
34.4.2 Traducteur	525
34.4.3 Relecture	525
<b>35 perldiag</b>	<b>526</b>
35.1 DESCRIPTION	526
35.2 TRADUCTION	558
35.2.1 Version	558
35.2.2 Traducteurs	558
35.2.3 Relecture	558
<b>36 perldebug</b>	<b>559</b>
36.1 DESCRIPTION	559
36.2 Le Débogueur Perl	559
36.2.1 Commandes du Débogueur	559
36.2.2 Options Configurables	564
36.2.3 entrées/sorties du débogueur	566
36.2.4 Débogage des instructions lors de la compilation	567
36.2.5 Personnalisation du Débogueur	567
36.2.6 Support de Readline	568
36.2.7 Support d'un Éditeur pour le Débogage	568
36.2.8 Le Profileur Perl	568
36.3 Débogage des expressions rationnelles	569
36.4 Débogage de l'usage de la mémoire	569
36.5 VOIR AUSSI	569
36.6 BUGS	569
36.7 TRADUCTION	569
36.7.1 Version	569
36.7.2 Traducteur	569
36.7.3 Relecture	569

<b>37 perlvar</b>	<b>570</b>
37.1 DESCRIPTION	570
37.1.1 Noms prédéfinis	570
37.1.2 Indicateurs d'erreur	585
37.1.3 Note technique sur la syntaxe de noms variables	586
37.2 BUGS	586
37.3 TRADUCTION	586
37.3.1 Version	586
37.3.2 Traducteur	586
37.3.3 Relecture	586
<b>38 perlre</b>	<b>587</b>
38.1 DESCRIPTION	587
38.1.1 Expressions rationnelles	588
38.1.2 Motifs étendus	592
38.1.3 Retour arrière	596
38.1.4 Version 8 des expressions rationnelles	599
38.1.5 AVERTISSEMENT concernant \1 et \$1	600
38.1.6 Répétition de motifs de reconnaissance de longueur nulle	600
38.1.7 Combinaison d'expressions rationnelles	601
38.1.8 Création de moteurs RE spécifiques (customisés)	602
38.2 BUGS	603
38.3 VOIR AUSSI	603
38.4 TRADUCTION	603
38.4.1 Version	603
38.4.2 Traducteur	603
38.4.3 Relecture	603
<b>39 perlref</b>	<b>604</b>
39.1 DESCRIPTION	604
39.1.1 OPÉRATEURS	604
39.1.2 SYNTAXE	605
39.1.3 SÉQUENCES D'ÉCHAPPEMENT	605
39.1.4 CLASSES DE CARACTÈRES	606
39.1.5 ANCRES	607
39.1.6 QUANTIFICATEURS	607
39.1.7 CONSTRUCTIONS ÉTENDUES	607
39.1.8 VARIABLES	607
39.1.9 FONCTIONS	608
39.1.10 TERMINOLOGIE	608
39.2 AUTEUR	608
39.3 VOIR AUSSI	608
39.4 REMERCIEMENTS	609
39.5 TRADUCTION	609
39.5.1 Version	609
39.5.2 Traducteur	609
39.5.3 Relecture	609



<b>40 perlref</b>	<b>610</b>
40.1 NOTE	610
40.2 DESCRIPTION	610
40.2.1 Créer des références	610
40.2.2 Utiliser des références	613
40.2.3 Références symboliques	615
40.2.4 Références pas-si-symboliques-que-ça	616
40.2.5 Pseudo-tables de hachage : utiliser un tableau comme table de hachage	617
40.2.6 Modèles de fonctions	618
40.3 AVERTISSEMENT	619
40.4 VOIR AUSSI	619
40.5 AUTEUR	619
40.6 TRADUCTION	619
40.6.1 Version	619
40.6.2 Traducteur	619
40.6.3 Relecture	619
<b>41 perlfm</b>	<b>620</b>
41.1 DESCRIPTION	620
41.1.1 Variables de formats	622
41.2 NOTES	622
41.2.1 Pied de page	623
41.2.2 Accéder aux formats de l'intérieur	624
41.3 MISE EN GARDE	624
41.4 TRADUCTION	624
41.4.1 Version	624
41.4.2 Traducteur	624
41.4.3 Relecture	624
<b>42 perlobj</b>	<b>625</b>
42.1 DESCRIPTION	625
42.1.1 Un objet est simplement une référence	625
42.1.2 Une classe est simplement un paquetage	627
42.1.3 Une méthode est simplement un sous-programme	627
42.1.4 Invocation de méthode	628
42.1.5 Syntaxe objet indirecte	629
42.1.6 Méthodes UNIVERSAL par défaut	630
42.1.7 Destructeurs	630
42.1.8 Résumé	631
42.1.9 Ramasse-miettes à deux phases	631
42.2 VOIR AUSSI	632
42.3 TRADUCTION	632
42.3.1 Version	632
42.3.2 Traducteur	632
42.3.3 Relecture	632

<b>43 perlite</b>	<b>633</b>
43.1 SYNOPSIS	633
43.2 DESCRIPTION	633
43.2.1 Les scalaires liés (par tie())	633
43.2.2 Les tableaux liés (par tie())	635
43.2.3 Les tables de hachage liées (par tie())	639
43.2.4 Les descripteurs de fichiers (filehandles) liés (par tie())	643
43.2.5 Les pièges du untie	645
43.3 VOIR AUSSI	647
43.4 BUGS	647
43.5 AUTEURS	647
43.6 TRADUCTION	647
43.6.1 Version	647
43.6.2 Traducteur	647
43.6.3 Relecture	647
<b>44 perlipc</b>	<b>648</b>
44.1 DESCRIPTION	648
44.2 Signaux	648
44.3 Tubes nommés	650
44.3.1 AVERTISSEMENT	651
44.4 Utilisation de open() pour la CIP	651
44.4.1 Handles de Fichiers	652
44.4.2 Processus en Arrière-Plan.	652
44.4.3 Dissociation Complète du Fils et de son Père	653
44.4.4 Ouvertures Sûres d'un Tube	653
44.4.5 Communication Bidirectionnelle avec un autre Processus	654
44.4.6 Communication Bidirectionnelle avec Vous-même	655
44.5 Sockets : Communication Client/Serveur	656
44.5.1 Terminateur de Ligne Internet	657
44.5.2 Clients et Serveurs TCP Internet	657
44.5.3 Clients et Serveurs TCP du Domaine Unix	660
44.6 Clients TCP avec IO::Socket	662
44.6.1 Un Client Simple	662
44.6.2 Un Client Webget	663
44.6.3 Client Interactif avec IO::Socket	663
44.7 Serveurs TCP avec IO::Socket	665
44.8 UDP : Transfert de Message	666
44.9 CIP du SysV	667
44.10NOTES	669
44.11BUGS	669
44.12VOIR AUSSI	669
44.13AUTEUR	669
44.14TRADUCTION	669
44.14.1 Version	669
44.14.2 Traducteur	670
44.14.3 Relecture	670

<b>45 perlnumber</b>	<b>671</b>
45.1 SYNOPSIS	671
45.2 DESCRIPTION	671
45.3 Stockage des nombres	671
45.4 Opérateurs et conversions numériques	672
45.5 Un avant-goût des opérations numériques en Perl	672
45.6 AUTEUR	673
45.7 VOIR AUSSI	673
45.8 TRADUCTION	673
45.8.1 Version	673
45.8.2 Traducteur	673
45.8.3 Relecture	673
<b>46 perlthrtut</b>	<b>674</b>
46.1 DESCRIPTION	674
46.2 État courant	674
46.3 Qu'est-ce que c'est qu'un thread, d'abord ?	674
46.4 Modèles de programmes utilisant les threads	674
46.4.1 Maître/Esclave	675
46.4.2 Équipe de travail	675
46.4.3 Travail à la chaîne	675
46.5 Implémentations des threads dans le système d'exploitation	675
46.6 De quelle sorte sont les threads de Perl ?	676
46.7 Modules réentrants ( <i>thread-safe</i> )	676
46.8 Bases des threads	677
46.8.1 Support de base pour les threads	677
46.8.2 Note à propos des exemples	677
46.8.3 Créer des threads	677
46.8.4 Rendre le contrôle	678
46.8.5 Attendre qu'un thread termine	679
46.8.6 Ignorer un thread	679
46.9 Threads et données	679
46.9.1 Données partagées et non partagées	680
46.9.2 Pièges des threads : <i>race conditions</i>	680
46.10 Synchronisation et contrôle	681
46.10.1 Contrôler l'accès : <code>lock()</code>	681
46.10.2 Un piège des threads : interblocages ( <i>deadlocks</i> )	682
46.10.3 Files d'attente ( <i>queues</i> ) : transmettre des données	683
46.10.4 Sémaphores : synchroniser les accès aux données	684
46.10.5 Sémaphores de base	684
46.10.6 Sémaphores avancés	684
46.10.7 <code>cond_wait()</code> et <code>cond_signal()</code>	685
46.11 Fonctions utiles générales	685
46.11.1 Dans quel thread suis-je ?	685
46.11.2 Identificateur de thread	685
46.11.3 Est-ce que ces threads sont les mêmes ?	686
46.11.4 Quels threads sont en train de tourner ?	686

46.12	Un exemple complet	686
46.13	Considérations de performance	687
46.14	Changements au niveau du processus	687
46.15	Réentrance des bibliothèques système	688
46.16	Conclusion	688
46.17	Bibliographie	688
46.17.1	Textes introductifs	688
46.17.2	Références liées aux systèmes d'exploitation	688
46.17.3	Autres références	688
46.18	Crédits	689
46.19	AUTEUR	689
46.20	Copyrights	689
46.21	Copyright	689
46.22	TRADUCTION	689
46.22.1	Version	689
46.22.2	Traducteur	689
46.22.3	Relecture	689
<b>47</b>	<b>perlport</b>	<b>690</b>
47.1	DESCRIPTION	690
47.2	PROBLÈMES	691
47.2.1	Les retours chariots	691
47.2.2	Stockage et taille des nombres	692
47.2.3	Fichiers	692
47.2.4	Interaction avec le système	693
47.2.5	Communication inter-processus (IPC)	693
47.2.6	Sous-programme externe (XS)	693
47.2.7	Modules Standard	694
47.2.8	Date et Heure	694
47.2.9	Jeux de caractère et encodage des caractères.	694
47.2.10	Internationalisation	694
47.2.11	Ressources Système	694
47.2.12	Sécurité	695
47.2.13	Style	695
47.3	Testeurs du CPAN	695
47.4	PLATEFORMES	695
47.4.1	Unix	695
47.4.2	DOS dérivés	696
47.4.3	Mac OS	696
47.4.4	VMS	697
47.4.5	Plateformes EBCDIC	699
47.4.6	Acorn RISC OS	699
47.4.7	Autres perls	701
47.5	IMPLÉMENTATION DES FONCTIONS	701
47.5.1	Liste Alphabétique des Fonctions Perl	701
47.6	AUTEURS / CONTRIBUTEURS	705
47.7	VERSION	706
47.8	TRADUCTION	706
47.8.1	Version	706
47.8.2	Traducteur	706
47.8.3	Relecture	706

<b>48 perllocale</b>	<b>707</b>
48.1 DESCRIPTION	707
48.2 PRÉ-REQUIS POUR L'UTILISATION DES "LOCALE"	707
48.3 UTILISATION DES "LOCALE"	708
48.3.1 La directive locale	708
48.3.2 La fonction setlocale	708
48.3.3 Trouver les "locale"	709
48.3.4 PROBLÈMES DE "LOCALE"	710
48.3.5 Résolution temporaire des problèmes de "locale"	710
48.3.6 Résolution permanente des problèmes de "locale"	710
48.3.7 Résolution permanente des problèmes de configuration système des "locale"	711
48.3.8 Configuration système des "locale"	711
48.3.9 La fonction localeconv	711
48.4 CATÉGORIES DE "LOCALE"	712
48.4.1 Catégorie LC_COLLATE: tri	712
48.4.2 Catégorie LC_CTYPE: type de caractères	713
48.4.3 Catégorie LC_NUMERIC: format numérique	713
48.4.4 Catégorie LC_MONETARY: format des valeurs monétaires	714
48.4.5 LC_TIME	714
48.4.6 Autres catégories	714
48.5 SÉCURITÉ	715
48.6 ENVIRONNEMENT	716
48.7 NOTES	717
48.7.1 Compatibilité	717
48.7.2 I18N:Collate obsolète	717
48.7.3 Impactes sur la vitesses des tris et sur l'utilisation de la mémoire	717
48.7.4 write() et LC_NUMERIC	717
48.7.5 Définitions de "locale" disponibles librement	718
48.7.6 I18n et I10n	718
48.7.7 Un standard imparfait	718
48.8 BUGS	718
48.8.1 Les systèmes bugués	718
48.9 VOIR AUSSI	718
48.10HISTORIQUE	718
48.11TRADUCTION	719
48.11.1 Version	719
48.11.2 Traducteur	719
48.11.3 Relecture	719

<b>49 perluniintro</b>	<b>720</b>
49.1 DESCRIPTION	720
49.1.1 Unicode	720
49.1.2 Le support d'Unicode en Perl	721
49.1.3 Le modèle Unicode de Perl	721
49.1.4 Unicode et EBCDIC	722
49.1.5 Créer de l'Unicode	722
49.1.6 Manipuler l'Unicode	723
49.1.7 Encodages natifs	723
49.1.8 Entrées/Sorties et Unicode	724
49.1.9 Afficher de l'Unicode sous forme de texte	725
49.1.10 Cas spéciaux	726
49.1.11 Sujets avancés	726
49.1.12 Divers	726
49.1.13 Questions/Réponses	727
49.1.14 Notation hexadécimale	728
49.1.15 Autres ressources	729
49.2 UNICODE DANS LES VERSIONS DE PERL PLUS ANCIENNES	729
49.3 REFERENCES	729
49.4 REMERCIEMENTS	729
49.5 AUTEUR, COPYRIGHT, ET LICENCE	730
49.6 TRADUCTION	730
49.6.1 Version	730
49.6.2 Traducteur	730
49.6.3 Relecture	730
<b>50 perlsec</b>	<b>731</b>
50.1 DESCRIPTION	731
50.1.1 Blanchiment et détection des données souillées	733
50.1.2 Options sur la ligne "#!"	733
50.1.3 Mode taint et @INC	734
50.1.4 Nettoyer votre PATH	734
50.1.5 Failles de sécurité	735
50.1.6 Protection de vos programmes	736
50.1.7 Unicode	736
50.1.8 Attaques par complexité algorithmique	736
50.2 VOIR AUSSI	737
50.3 TRADUCTION	737
50.3.1 Version	737
50.3.2 Traducteur	737
50.3.3 Relecture	737

<b>51 perlmod</b>	<b>738</b>
51.1 DESCRIPTION	738
51.1.1 Paquetages	738
51.1.2 Tables de symboles	739
51.1.3 Constructeurs et Destructeurs de paquetage	740
51.1.4 Classes Perl	740
51.1.5 Modules Perl	740
51.2 VOIR AUSSI	743
51.3 TRADUCTION	743
51.3.1 Version	743
51.3.2 Traducteur	743
51.3.3 Relecture	743
<b>52 perlmodlib</b>	<b>744</b>
52.1 DESCRIPTION	744
52.2 LA LIBRAIRIE DE MODULES PERL	744
52.2.1 Pragmatic Modules	744
52.2.2 Standard Modules	745
52.2.3 Extension de Modules	749
52.3 CPAN	749
52.4 Modules: Création, Utilisation, et Abus	751
52.4.1 Directives pour la création de modules	751
52.4.2 Directives pour convertir des bibliothèques Perl 4 en modules	754
52.4.3 Directives pour réutiliser le code d'application	755
52.5 NOTE	755
52.6 TRADUCTION	755
52.6.1 Version	755
52.6.2 Traducteur	755
52.6.3 Relecture	755
<b>53 perlmodinstall</b>	<b>756</b>
53.1 DESCRIPTION	756
53.1.1 PRÉAMBULE	756
53.2 OYEZ !	759
53.3 AUTEUR	759
53.4 COPYRIGHT	759
53.5 TRADUCTION	760
53.5.1 Version	760
53.5.2 Traducteur	760
53.5.3 Relecture	760
<b>54 perlutil</b>	<b>761</b>
54.1 DESCRIPTION	761
54.1.1 DOCUMENTATION	761
54.1.2 CONVERTISSEURS	762
54.1.3 Administration	762
54.1.4 Développement	762
54.1.5 VOIR AUSSI	763
54.2 TRADUCTION	763
54.2.1 Version	763
54.2.2 Traducteur	763
54.2.3 Relecture	763

<b>IV</b>	<b>Implémentation et interface avec le langage C</b>	<b>764</b>
<b>55</b>	<b>perlembed</b>	<b>765</b>
55.1	DESCRIPTION	765
55.1.1	PRÉAMBULE	765
55.1.2	SOMMAIRE	765
55.1.3	Compiler votre programme C	766
55.1.4	Ajouter un interpréteur Perl à votre programme C	767
55.1.5	Appeler un sous-programme Perl à partir de votre programme C	767
55.1.6	Évaluer un expression Perl à partir de votre programme C	768
55.1.7	Effectuer des recherches de motifs et des substitutions à partir de votre programme C	769
55.1.8	Trifouiller la pile Perl à partir de votre programme C	772
55.1.9	Maintenir un interpréteur persistant	774
55.1.10	Maintenir de multiples instances d'interpréteur	776
55.1.11	Utiliser des modules Perl utilisant les bibliothèques C, à partir de votre programme C	777
55.2	Intégrer du Perl sous Win32	779
55.3	MORALITE	779
55.4	AUTEUR	779
55.5	COPYRIGHT ORIGINAL	779
55.6	TRADUCTION	780
55.6.1	Version	780
55.6.2	Traducteurs	780
55.6.3	Relecture	780
<b>56</b>	<b>perldebguts</b>	<b>781</b>
56.1	DESCRIPTION	781
56.2	Éléments Internes du Débogueur	781
56.2.1	Écrire Votre Propre Débogueur	782
56.3	Exemples de Listages des Frames	783
56.4	Débogage des expressions rationnelles	785
56.4.1	Trace lors de la compilation	785
56.4.2	Types de noeuds	787
56.4.3	Sortie lors de l'exécution	789
56.5	Débogage de l'utilisation de la mémoire par Perl	789
56.5.1	Utilisation de \$ENV{PERL_DEBUG_MSTATS}	790
56.5.2	Exemple d'utilisation de l'option <b>-DL</b>	791
56.5.3	Détails sur <b>-DL</b>	792
56.5.4	Limitations des statistiques <b>-DL</b>	792
56.6	VOIR AUSSI	792
56.7	TRADUCTION	793
56.7.1	Version	793
56.7.2	Traducteur	793
56.7.3	Relecture	793



<b>57 perlXStut</b>	<b>794</b>
57.1 DESCRIPTION	794
57.1.1 VERSION DE CE DOCUMENT	794
57.1.2 DYNAMIQUE / STATIQUE	794
57.1.3 EXEMPLE 1	795
57.1.4 EXEMPLE 2	797
57.1.5 QUE S'EST-IL DONC PASSE ?	797
57.1.6 ECRIRE DE BONS SCRIPTS DE TEST	798
57.1.7 EXEMPLE 3	798
57.1.8 QU'Y A-T-IL DE NOUVEAU ICI ?	799
57.1.9 PARAMETRES D'ENTREE ET DE SORTIE	799
57.1.10 LE COMPILATEUR XSUBPP	799
57.1.11 LE FICHER TYPEMAP	799
57.1.12 AVERTISSEMENT	800
57.1.13 EXEMPLE 4	800
57.1.14 QUE S'EST-IL PASSE ICI ?	802
57.1.15 LE PASSAGE D'ARGUMENTS A XSUBPP	802
57.1.16 LA PILE DES ARGUMENTS	803
57.1.17 ETENDRE VOTRE EXTENSION	803
57.1.18 DOCUMENTATION DE VOTRE EXTENSION	803
57.1.19 INSTALLATION DE VOTRE EXTENSION	803
57.1.20 VOIR AUSSI	803
57.2 AUTEUR	804
57.2.1 Date de dernière modification	804
57.3 TRADUCTION	804
57.3.1 Version	804
57.3.2 Traducteur	804
57.3.3 Relecture	804
<b>58 perlxs</b>	<b>805</b>
58.1 DESCRIPTION	805
58.1.1 Introduction	805
58.1.2 Mise en route	805
58.1.3 Anatomie d'une XSUB	806
58.1.4 La pile des arguments	807
58.1.5 La variable RETVAL	807
58.1.6 Le mot-clé MODULE	807
58.1.7 Le mot-clé PACKAGE	808
58.1.8 Le mot-clé PREFIX	808
58.1.9 Le mot-clé OUTPUT	808
58.1.10 Le mot-clé CODE	809
58.1.11 Le mot-clé INIT	809
58.1.12 Le mot-clé NO_INIT	809
58.1.13 Initialisation des arguments de fonction	810
58.1.14 Valeurs de paramètres par défaut	810
58.1.15 Le mot-clé PREINIT	811
58.1.16 Le mot-clé SCOPE	811

58.1.17	Le mot-clé INPUT	812
58.1.18	Listes de paramètres de longueur variable	812
58.1.19	Le mot-clé C_ARGS	813
58.1.20	Le mot-clé PPCODE	813
58.1.21	Renvoyer undef et des listes vides	814
58.1.22	Le mot-clé REQUIRE	815
58.1.23	Le mot-clé CLEANUP	815
58.1.24	Le mot-clé BOOT	815
58.1.25	Le mot-clé VERSIONCHECK	815
58.1.26	Le mot-clé PROTOTYPES	816
58.1.27	Le mot-clé PROTOTYPE	816
58.1.28	Le mot-clé ALIAS	816
58.1.29	Le mot-clé INTERFACE	817
58.1.30	Le mot-clé INTERFACE_MACRO	817
58.1.31	Le mot-clé INCLUDE	818
58.1.32	Le mot-clé CASE	818
58.1.33	L'opérateur unaire &	819
58.1.34	Insérer des commentaires et des directives de pré-processeur C	819
58.1.35	Utiliser XS avec C++	819
58.1.36	Méthode de construction d'interfaces	821
58.1.37	Objets Perl et structures C	821
58.1.38	Le typemap	822
58.2	EXEMPLES	823
58.3	VERSION DE XS	824
58.4	AUTEUR	824
58.5	TRADUCTION	824
58.5.1	Version	824
58.5.2	Traducteur	824
58.5.3	Relecture	824
<b>59</b>	<b>perlintern</b>	<b>825</b>
59.1	DESCRIPTION	825
59.2	AUTEURS	825
59.3	VOIR AUSSI	825
59.4	TRADUCTION	825
59.4.1	Version	825
59.4.2	Traducteur	825
59.4.3	Relecture	825
<b>60</b>	<b>perlpio</b>	<b>826</b>
60.1	SYNOPSIS	826
60.2	DESCRIPTION	827
60.2.1	Co-existence avec stdio	828
60.3	TRADUCTION	829
60.3.1	Version	829
60.3.2	Traducteurs	829
60.3.3	Relecture	829

<b>V Divers</b>	<b>830</b>
<b>61 perlbook</b>	<b>831</b>
61.1 DESCRIPTION	831
61.2 TRADUCTION	831
61.2.1 Version	831
61.2.2 Traducteur	831
61.2.3 Relecture	831
<b>62 perlhst</b>	<b>832</b>
62.1 DESCRIPTION	832
62.2 INTRODUCTION	832
62.3 LES GARDIENS DE LA CITROUILLE	832
62.3.1 LA CITROUILLE ?	832
62.4 LES ARCHIVES	833
62.4.1 TAILLES DE VERSIONS CHOISIES	838
62.4.2 TAILLES DE PATCH CHOISIES	841
62.5 LES GARDIENS DES ARCHIVES	842
62.6 TRADUCTION	842
62.6.1 Version	842
62.6.2 Traducteur	842
62.6.3 Relecture	842
<b>63 perlartistic</b>	<b>843</b>
63.1 SYNOPSIS	843
63.2 DESCRIPTION	843
63.3 Avertissement du traducteur	843
63.4 La « licence artistique »	843
63.4.1 Préambule	843
63.4.2 Définitions	843
63.4.3 Conditions de modification et de distribution	844
63.5 TRADUCTION	845
63.5.1 Version	845
63.5.2 Traducteur	845
63.5.3 Relecture	845
63.6 The "Artistic License"	845
63.6.1 Preamble	845
63.6.2 Definitions	846
63.6.3 Conditions	846
<b>VI Spécificités pour certaines langues</b>	<b>848</b>
<b>VII Spécificités pour certaines plates-formes</b>	<b>849</b>