Ecole Nationale de l'Aviation Civile
Département Mathématiques et Informatique

LE LANGAGE C

A. DANCEL

Envoyez SVP vos commentaires et corrections à :
dancel@eis.enac.dgac.fr

Table des Matières

Liste des Figures

Chapitre 1

Généralités

1.1 Historique

- Le langage C est né en 1972 dans les laboratoires de la Bell Telephone (AT&T) des travaux de Brian Kernighan et Dennis Ritchie.
- Il a été conçu à l'origine pour l'écriture du système d'exploitation UNIX (90-95% du noyau écrit en C) et s'est vite imposé comme le langage de programmation sous UNIX.
- Très inspiré des langages BCPL (Martin Richard) et B (Ken Thompson), il se présente comme un "super-assembleur" ou "assembleur portable".

  En fait c'est un compromis entre un langage de haut niveau (Pascal, Ada ...) et un langage de bas niveau (assembleur).
- Il a été normalisé en 1989 par le comité X3J11 de l'American National Standards Institute (ANSI).

1.2 Caractéristiques

- **Langage structuré** conçu pour traiter les tâches d'un programme en les mettant dans des blocs.
- produit des **programmes efficaces** : il possède les mêmes possibilités de contrôle de la machine que l'assembleur.
- **Déclaratif** : tout objet C doit être déclaré avant d'être utilisé.
- **Format libre** : la mise en page des divers composants d'un programme est totalement libre.

  Cette possibilité doit être exploitée pour rendre les programmes lisibles.
- **Modulaire** : une application pourra être découpée en modules qui pourront être compilés séparément.

  Un ensemble de programmes déjà opérationnels pourra être réuni dans une **librairie**. Cette aptitude permet à C de se développer de lui même.

- **Souple** : peu de vérifications et d'interdits ...
- **Transportable** : les entrées/sorties sont réunies dans une librairie externe au langage.
- Sa spécificité vient de son traitement des pointeurs et à son aptitude à générer un code compact et rapide.

1.3 Forme générale d'un programme C

1.3.1 Structure d'un programme C

Un programme C est composé de :

- **Directives du préprocesseur** : elles permettent d'effectuer des manipulations sur le texte du programme source avant la compilation (inclusion de fichiers, substitutions, macros, compilation conditionnelle).

  Une directive du préprocesseur est une ligne de programme source commençant par le caractère dièse (#).

  Le préprocesseur (`/lib/cpp`) est appelé automatiquement par la commande `/bin/cc`.

- **Déclarations/définitions** :

  Déclaration : la déclaration d'un objet C donne simplement ses caractéristiques au compilateur et ne génère aucun code.

  Définition : la définition d'un objet C déclare cet objet et crée effectivement cet objet.

- **Fonctions** : Ce sont des sous-programmes dont les instructions vont définir un traitement sur des variables.

  Elles peuvent retourner une valeur à la fonction appelante.

  Le programme principal est une fonction dont le nom doit impérativement être `main`.

  Les fonctions ne peuvent pas être imbriquées.

- **Des commentaires** : éliminés par le préprocesseur, ce sont des textes compris entre `/*` et `*/`. On ne doit pas les imbriquer et ils peuvent apparaître en tout point d'un programme (sauf dans une constante de type chaîne de caractères ou caractère).

  Pour ignorer une partie de programme il est préférable d'utiliser une directive du préprocesseur (`#if 0 ... #endif`)

1.3.2 Structure d'une fonction

Une fonction est un bloc de code d'une ou plusieurs instructions qui peut renvoyer une valeur à l'expression qui l'utilise.

La forme générale d'une fonction est :

```
[classe] [type] nom( [liste_de_parametres_formels] )
  declaration_des_parametres_formels
  bloc_de_la_fonction
```

1.3.3 Structure d'un bloc

- Un bloc est une suite d'instructions élémentaires délimitées par des accolades { et }
- Un bloc peut commencer par des déclarations/définitions d'objets qui seront locaux à ce bloc.
- Un bloc peut contenir un ou plusieurs blocs inclus.

1.3.4 Structure d'une instruction élémentaire

Une instruction élémentaire est une expression terminée par le caractère ; (point virgule)

1.4 Règles d'écriture des programmes C

Afin d'écrire des programmes C lisibles, il est important de respecter un certain nombre de règles de présentation :

- ne jamais placer plusieurs instructions sur une même ligne
- utiliser des identificateurs significatifs
- grace à l'indentation[1] des lignes, on fera ressortir la structure syntaxique du programme. Les valeurs de décalage les plus utilisées sont de 2, 4 ou 8 espaces
- on laissera une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions
- une accolade fermante est seule sur une ligne[2] et fait référence, par sa position horizontale, au début du bloc qu'elle ferme
- aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces
- il est nécessaire de commenter les listings. Eviter les commentaires triviaux

[1] décalage horizontal des lignes

[2] à l'exception de l'accolade fermante du bloc de la structure `do ... while`

```
/* *************************
   Programme de demonstration
   *************************
*/

#include <stdio.h>  /* directives au preprocesseur */
#define DEBUT -10
#define FIN 10
#define MSG "Programme de demonstration\n";

int carre(int x);   /* declaration des fonctions */
int cube(int x);

main()              /* programme principal */
{                   /* debut du bloc de la fonction main */
  int i;            /* definition des variables locales */

  printf(MSG);
  for ( i = DEBUT; i <= FIN ; i++ ) {
     printf("%d   carre: %d   cube: %d\n", i
                                        , carre(i)
                                        , cube(i) );

  }                 /* fin du bloc for  */
  return 0;
}                   /* fin du bloc de la fonction main */


int cube(int x) {   /* definition de la fonction cube */
   return x * carre(x);
}

int carre(int x) {  /* definition de la fonction carre */
   return x * x;
}
```

Figure 1.1: Exemple de programme C

```
/* westley.c */

#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
                  _-_-_-_
              _-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_
              _-_-_-_-_-_-_-_-_
                  _-_-_-_
}
```

Compilation et execution :

```
apollo26:[~]> cc -o westley westley.c
apollo26:[~]> westley
3.141
apollo26:[~]>
```

Figure 1.2: Programme C illisible (International Obfuscated C Code Contest 1988)

Chapitre 2

Constantes, variables et types de base

2.1 Identificateurs

Les identificateurs nomment les objets C (fonctions, variables ...)

- C'est une suite de lettres ou de chiffres.
- Le premier caractère est obligatoirement une lettre.
- Le caractère _ (souligné) est considéré comme une lettre.
- Le C distingue les minuscules des majuscules. Exemple : `carlu Carlu CarLu CARLU` sont des identificateurs valides et tous différents.
- La longueur de l'identificateur dépend de l'implémentation. La norme ANSI prévoit qu'au moins les 31 premiers caractères soient significatifs pour le compilateur. L'éditeur de liens peut limiter le nombre de caractères significatifs des identificateurs à un nombre plus petit.
- Un identificateur ne peut pas être un mot réservé du langage :

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Figure 2.1: Mots réservés

Recommandations :

- Utiliser des identificateurs significatifs.
- Réserver l'usage des identificateurs en majuscules pour le préprocesseur.
- Réserver l'usage des identificateurs commençant par le caractère _ (souligné) à l'usage du compilateur.

2.2 Types de base

C'est à partir de ces types de base que l'on peut construire tous les autres types, dits types dérivés (tableaux, structures, unions ...).

2.2.1 Caractère

Le type `char` représente un caractère parmi l'ensemble des caractères du jeu de caractères de la machine. Le code des caractères utilisés n'est pas défini par le langage. Dans la grande majorité des cas c'est le code ASCII.

- Taille : 1 octet
- intervalle : -128 à +127 ou 0 à 255 suivant les machines

Il peut être :

- `signed char` : intervalle de -128 à 127.
- `unsigned char` : intervalle de 0 à 255.

Ces 2 derniers types rendent les programmes plus portables.

2.2.2 Entier

Le type `int` permet de représenter les nombres entiers. Il correspond à un mot machine. Sa taille dépend donc de la taille du processeur utilisé. Elle est en général de 16 ou 32 bits.

Il peut être qualifié par :

- `short` ou `long` : spécifie la taille de l'emplacement mémoire utilisé.
- `unsigned` : entier positif.
- `signed` : entier signé.

`int` est facultatif dès qu'il est précédé de `short`, `long` ou `unsigned`

La seule chose que précise la norme ANSI est que :
`taille( short ) <= taille( int ) <= taille( long )`
Le type `int` est signé par défaut.

2.2.3 Les réels

Ils permettent de représenter un nombre en virgule flottante. Ils se divisent en :

- `float` : pour les nombres en virgule flottante simple précision
- `double` : pour les nombres en virgule flottante double précision
- `long double` : pour les nombres en virgule flottante à précision étendue.

Les fichiers d'en-tête `limits.h` (c.f. page 127) et `float.h` (c.f. page 128) contiennent des constantes symboliques qui précisent les tailles et les valeurs limites des entiers et des flottants.

2.2.4 Les différents types sous HP C.

Type	Bits	Min.	Max.
char	8	-128	127
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32768	32767
unsigned short	16	0	65535
int	32	-2147483648	2147483647
unsigned	32	0	4294967295
long	32	-2147483648	2147483647
unsigned long	32	0	4294967295
float	32	1.17 E-38	3.4 E+38
double	64	2.22 E-308	1.7 E+308
long double	128	3.36 E-4932	1.1 E+4932

Figure 2.2: Taille des types sous HP C

2.3 Les constantes

Chaque constante a une valeur et un type. Les expressions constantes sont évaluées à la compilation ; ainsi l'expression constante `1 + 2 * 3` est interprétée comme `7`.

2.3.1 Constantes entières

Les constantes entières peuvent être exprimées sous les formes suivantes :

- Décimale : exemple : `1234`
- Octale : le premier chiffre est obligatoirement un zéro. Exemple : `0177`

- Hexadécimale : le nombre commence par `0x` ou `0X`. On peut utiliser les lettres minuscules ou majuscules pour les chiffres hexadécimaux. Exemple : `0x1Bf  0XF2a`

Les entiers trop grands pour tenir dans un `int` seront considérés comme du type `long`.
On peut imposer à une constante entière d'être du type :

- `long` : en la suffixant **l** ou **L** à la fin (il est préférable d'utiliser le L majuscule afin d'éviter la confusion entre le l minuscule et le chiffre 1). Exemple : `123L`

  Exemple : pourquoi cette instruction affiche `3` au lieu de `32` ?

```
printf("11 + 21 = %d\n", 1l + 2l );
```

- `unsigned` : en la suffixant **u** or **U** à la fin. Exemple : `3456U` , `78UL` (`unsigned long`)

2.3.2 Constantes réelles

Les constantes réelles sont, par défaut, de type `double`.
Elles peuvent être exprimées sous les formes suivantes :

- Décimale : exemple : `1234.56  4.  -.3456`

- Scientifique : exemple : `1.23e-12  12.3456E4`

Elles peuvent aussi inclure un suffixe :

- **f** ou **F** : impose le type `float` à la constante. Exemple : 1.23f

- **l** ou **L** : impose le type `long double`. Exemple : 123.4567E45L

2.3.3 Constantes caractères

Une constante de type caractère est écrite entre apostrophes.
Elles peuvent être exprimées sous les formes suivantes :

- Normale : `'a'`

- Octale : `'\007'`

- Hexadécimale : `'\x1b'`

- Symbolique : (c.f. figure 2.3)

La valeur d'une constante caractère est égale à la valeur numérique du caractère dans le code caractère de la machine.

NOTATION	CODE ASCII (hexa)	SIGNIFICATION
'\n'	0x0A	saut de ligne
'\t'	0x09	tabulation horizontale
'\v'	0x0B	tabulation verticale
'\a'	0x07	beep
'\b'	0x08	retour arrière
'\r'	0x0D	retour chariot
'\f'	0x0C	saut de page
'\\'	0x5C	backslash
'\''	0x2C	simple quote
'\"'	0x22	double quote
'\?'	0x3F	point d'interrogation

Figure 2.3: Forme symbolique des constantes caractères

2.3.4 Constantes chaînes de caractères

Elles sont constituées d'une séquence de caractères, et délimitées par le caractère " (guillemet).

Toutes les notations de caractères (normale, octale, hexadécimale ou symbolique) sont utilisables dans les chaînes.

Exemples :

```
"Le langage C"  "Bonjour\n"  "\"Hello\""   "Anti-slash : \\"   ""
```

- Un caractère nul ('\0') est ajouté automatiquement à la fin de la chaîne.
- Les constantes chaînes de caractères sont stockées en zone de mémoire permanente.
- Elles sont considérées comme des tableaux de caractères dont la dimension est celle de chaîne plus 1 (pour le caractère nul placé à la fin du tableau).
- Une chaîne peut tenir sur plusieurs lignes :

  Exemples :

```
       "Ceci est une constante chaine\
        sur 2 lignes"
ou
       "Ceci est une autre forme"
       "de chaine sur 2 lignes"
```

- Il n'existe pas d'opérateurs sur les chaînes de caractères, mais il existe une librairie de fonctions de manipulations de chaînes qui permet d'effectuer toutes les opérations courantes. (c.f. page 128)

2.4 Définition d'une variable simple

Les variables doivent obligatoirement être déclarées avant d'être utilisées.

La syntaxe générale d'une définition ou déclaration de variable est de la forme :

```
classe modificateur type identificateur ;
```

- **classe** : (facultative) elle détermine le mode de stockage, la durée de vie et la visibilité de la variable.

  Elle peut prendre une des valeurs suivantes :

  `auto extern register static`

- **modificateur** : (facultatif) il peut prendre une des valeurs suivantes :

  `const volatile`

- **type** : c'est un type de base ou un type utilisateur.

Exemple :

```
int   x;                  /* x est un entier */
float x1, x2;             /* x1 et x2 sont des entiers */
static double delta;      /* delta est un double de classe static */
extern unsigned char rep;
```

2.5 Définition d'une variable tableau

Un tableau est une variable composée d'une suite séquentielle d'éléments de même type.

La syntaxe générale d'une définition de variable tableau à une dimension est de la forme :

```
classe modificateur type identificateur[exp_const_entiere] ;
```

La `classe`, le `modificateur` et le `type` sont les mêmes que pour les variables simples.

Le nombre d'éléments du tableau est déclaré entre les crochets par une expression constante entière évaluable par le compilateur. (Une variable ne peut donc pas apparaître dans l'expression définissant la taille du tableau)

Exemples :

```
int tab[10];            /* tab est un tableau de 10 entiers */
static float x[2*50+2]; /* x est un tableau de 102 reels    */
char ch1[20], ch2[40];
```

- Pour un tableau de dimension `n`, les indices valides vont de `0` à `n-1`.

- Pour une déclaration de tableau ou lorsque le tableau est initialisé lors de sa définition, la dimension du tableau peut être omise.
- Il est recommandé de donner un nom à la constante qui indique le nombre d'éléments du tableau.

  Exemple :

```
#define NBRE 50

int tab[NBRE];
```

Tableau multi-indicés

Un tableau multidimensionnel est un tableau dont les éléments sont eux-mêmes des tableaux.

Le nombre successif d'éléments de chaque dimension est déclaré entre crochets. Les éléments sont rangés consécutivement en mémoire "ligne par ligne".

Exemple :

```
int ecr[25][80];
```

Les éléments du tableau `ecr` sont rangés en mémoire dans l'ordre :

```
ecr[0][0] ecr[0][1] ... ecr[0][79] ecr[1][0] ... ecr[24][79]
```

ATTENTION :

- Il n'y a pas en C de vérification de débordement de tableau à l'exécution.

  Ainsi, l'instruction `ecr[0][80]='*';` écrase le contenu de l'élément `ecr[1][0]`. Ceci constitue une erreur courante, difficile à détecter.

  D'autre part, l'instruction `ecr[25][79]='*';` provoque l'écriture du caractère `*` en dehors de la zone allouée au tableau.
- La notation `ecr[0, 5]` ne désigne pas (comme en Pascal) `ecr[0][5]` mais l'élément `ecr[5]`. Cette erreur ne sera pas détectée par le compilateur (car la virgule est un opérateur).

2.6 Classes de stockage et modificateurs

2.6.1 Classe de stockage d'une variable

La classe de stockage d'une variable permet de préciser la durée de vie (permanente ou temporaire) et la visibilité de la variable (partie du programme où cette variable est utilisable).

Variables globales

Ce sont des variables déclarées ou définies en dehors de tout bloc (niveau externe). Elles ont une **durée de vie permanente**. Elles sont allouées au début de l'exécution du programme et ne sont libérées qu'à la fin de l'exécution du programme.

Par défaut elles sont visibles dans d'autres modules.

La classe **extern** permet de déclarer des variables qui sont définies (au niveau externe) dans un autre module.
La classe **static** limite la visibilité au module des variables globales.

Variables locales

Ce sont des variables définies à l'intérieur d'un bloc. La visibilité de telles variables s'étend uniquement au bloc et aux blocs éventuellement contenus.

Elles peuvent être de classe :

- **auto** : (par défaut) elles sont stockées dans la pile, leur durée de vie est temporaire (limitée à la durée de vie d'exécution du bloc). Elles sont allouées à l'entrée du bloc et libérées à la sortie du bloc.

- **static** : elles sont stockées en zone de donnée statique, leur durée de vie est permanente.

  Si une variable de cette classe est initialisée, cette initialisation est réalisée à la compilation du programme.

  La classe `static` permet de préserver la valeur de la variable entre les appels successifs à cette fonction (variable rémanente).

  Exemple :

```
f1() {
  static int i = 0;

  printf("%d\n", i);
  i = i + 1;
}
```

  La variable locale `i` (de classe `static`) est initialisée à la compilation avec la valeur 0. Le premier appel à la fonction `f1` affiche `0`, le deuxième appel affiche `1` ...

- **register** : seule une variable de type `char`, `int` ou `long` peut utiliser cette classe, qui demande au compilateur d'utiliser si possible un registre du microprocesseur afin d'optimiser l'accès à celle ci. Sa durée de vie est temporaire.

  Si l'allocation dans un registre n'est pas possible, la variable est considérée de classe `auto`.

Exemple :

```
/* classe, visibilite et masquage des variables */
int  an;
static char rep;
extern int  mode;

main() {
  char str[80];
  register int i;
  static   int jour;

  ...

       {  int i;

          ...
       }

  ...
}
```

- `int an;` : définition d'une variable globale visible dans tous les modules de l'application.
- `static char rep` : définition d'une variable globale visible uniquement dans ce module.
- `extern int  mode;` : déclaration d'une variable entière. Celle ci est définie dans une autre module par : `int  mode` (au niveau global de cet autre module).
- `char str[80];` : définition d'une variable tableau locale de classe `auto`.
- `register int i;` : définition d'une variable locale de classe `register`.
- `static int jour;` : définition d'une variable locale de classe `static`.
- `int i;` : définition d'une variable locale qui masque la variable locale `i` définie dans le bloc supérieur.

  Les variables locales à un bloc masquent les variables de mêmes noms définies à l'extérieur de ce bloc. Les variables masquées conservent leur valeur et redeviennent visibles à la sortie du bloc.

  L'abondance de masquage peut nuire à la clarté d'un programme.

2.6.2 Modificateurs de type

Il peut prendre une des valeurs suivantes :

- `const` : informe le compilateur que cette variable ne pourra pas faire l'objet d'une modification de son contenu. Celle ci n'est accessible qu'en lecture.

  Exemple :

```
const int ANNEE = 1992;
```

- `volatile` : informe le compilateur de ne pas faire d'optimisation sur cette variable, parce que cette variable sera modifiée probablement par une cause extérieure (interruption).

2.7 Initialisation des variables

2.7.1 Initialisation explicite

Les variables peuvent être initialisées lors de leur définition.

Exemple :

```
int i , j = 5;   /* initialise la variable j a 5 */
char c = 'R';
char tab1[3] = { 'a' , 'b' , 'c' };
char tab2[ ] = { 'a' , 'b' , 'c' };
```

Le compilateur détermine pour `tab2` le nombre d'éléments en fonction du nombre d'initialisateurs.

```
int tab3[3][2] = { {1,2},{3,4},{5,6}};
int tab4[3][2] = { 1, 2, 3, 4, 5, 6}; /* pareil que  tab3   */
int tab5[4]    = { 1, 2 };   /* tab5[2] = 0  et  tab5[3]=0  */
```

S'il y a moins d'initialisateurs que d'éléments déclarés, les éléments non initialisés le sont implicitement à `0`.

Remarque :

- En compilation classique(`K&R`), une variable locale tableau ne pourra être initialisée que si elle est de classe `static`.
- D'une manière générale, une variable peut être initialisée avec la valeur d'une expression :

Exemples :

```
short a = 0X01 << 3;
int   b = 2*a + 3;
int   nb = atoi(argv[1]);
```

2.7.2 Initialisation implicite

Elle dépend de la classe :

- classe `extern` ou `static` : ces variables sont permanentes. En l'absence d'initialisation explicite, elles sont initialisées à **0**.
- `auto` ou `register` : ces variables ne sont pas initialisées par défaut. Elles contiennent donc une valeur indéfinie.

Chapitre 3

Les expressions C

3.1 Généralités

- Une expression est obtenue à partir d'opérateurs, de variables, de constantes, de fonctions et de parenthèses.
- Une expression retourne toujours un résultat : la valeur de l'expression, qui a un type.
- Cette valeur retournée peut être affectée, testée ou utilisée comme opérande d'une autre expression.

  Par exemple, les expressions :

  ```
  5     a     f1()     a + b     3 * ( f1() + 5 )
  ```

  sont des expressions valides.

3.2 Les opérateurs arithmétiques

Tous ces opérateurs (c.f. figure 3.1) acceptent des opérandes pouvant être des entiers, des caractères ou des réels (sauf l'opérateur modulo).

Remarques :

- `%` : modulo : reste de la division entière (entre entiers seulement). Si un des opérandes est négatif, le signe du reste dépend de l'implémentation. En général il est de même signe que le dividende :

  Exemple :

  ```
  printf("%d\n", -3 % 2 );   /* affichage de -1 */
  printf("%d\n", 3 % -2 );   /* affichage de +1 */
  ```

- Il n'existe pas d'opérateur d'exponentiation.

opérateur	opération
-	moins unaire
+	plus unaire
*	multiplication
/	division
+	addition
-	soustraction
%	modulo : reste de la division entière (entre entiers)

Figure 3.1: Opérateurs arithmétiques

3.3 Opérateurs de manipulation de bits

Ces opérateurs (c.f. figure 3.2) opèrent bit à bit et s'appliquent à des opérandes de type entier (ou caractère) et de préférence `unsigned` (sinon ils risquent de modifier le bit de signe).

Ils procurent des possibilités de manipulation de bas-niveau de valeurs, traditionnellement réservées à la programmation en langage assembleur.

opérateur	opération
~	négation bit à bit (unaire)
<<	décalage à gauche
>>	décalage à droite
&	ET bit à bit
|	OU (inclusif) bit à bit
^	OU exclusif bit à bit

Figure 3.2: Opérateurs de manipulation de bits

- `~` : négation bit à bit (opérateur unaire). Il inverse un à un tous les bits de son unique opérande.

  Exemple :

  `~ 0x5F` est une expression qui vaut : `160` (`0xA0`)

  ```
  0x5F      0 1 0 1  1 1 1 1

  ~ 0x5F    1 0 1 0  0 0 0 0   ->  0xA0  ->  160
  ```

- `<<` : décalage à gauche.

  `a << 3` décale de 3 rangs à gauche la valeur contenue dans `a`. Les bits sortants à gauche sont perdus et des 0 sont introduits à droite. Cet opérateur permet d'effectuer des multiplications d'entiers par des puissances de 2. Si la variable est signée, le bit de signe est conservé.

- `>>` : décalage à droite.

  `a >> 3` décale de 3 rangs à droite la valeur contenue dans `a`. Les bits sortants à droite sont perdus et des 0 sont introduits à gauche. Cet opérateur permet d'effectuer des divisions d'entiers par des puissances de 2. Si la variable est signée, le bit de signe est conservé et propagé.

  Exemple :

```
int i = -100;

printf("%d\n", i >> 2 );   /* affichage de -25 */
printf("%d\n", i << 2 );   /* affichage de -400 */
```

- `&` : ET bit à bit entre les valeurs de 2 expressions.

  `a & b` chaque bit du résultat vaut 1 si les bits de même rang de `a` et `b` valent 1, 0 sinon.

- `|` : OU (inclusif) bit à bit entre les valeurs de 2 expressions.

  `a | b` chaque bit du résultat vaut 1 si les bits de même rang de `a` ou `b` valent 1, 0 sinon.

- `^` : OU exclusif bit à bit entre les valeurs de 2 expressions.

  `a ^ b` chaque bit du résultat vaut 1 si les bits de même rang de `a` et `b` sont différents, 0 sinon.

op1	op2	op1 & op2	op1 | op2	op1 ^ op2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Figure 3.3: Table de vérité des opérateurs bit à bit

Exemple :
`0xB6 & 0x53` est une expression qui vaut `18` (`0x12`) parce que :

```
  1 0 1 1  0 1 1 0          0xB6
&                         &
  0 1 0 1  0 0 1 1          0x53
  ---------------           -----
  0 0 0 1  0 0 1 0     ->   0x12    ->  18
```

3.4 Les opérateurs d'affectation

3.4.1 Valeur à gauche (lvalue)

Une valeur à gauche (lvalue [1]) est une expression qui peut apparaître à gauche de l'opérateur d'affectation. Une lvalue possède une adresse en mémoire.

Exemple : soit les déclarations suivantes :

```
int  alpha , beta ;
char tab[10];

5           /* la constante 5 n'est pas une lvalue */
alpha       /* alpha est une lvalue */
alpha+beta  /* alpha+beta n'est pas une lvalue */
tab[1]      /* tab[1] est une lvalue */
tab         /* n'est pas une lvalue. C'est un pointeur constant */
```

3.4.2 L'opérateur d'affectation

En C, l'affectation est un opérateur comme les autres. L'objet à gauche du `=` (égal) se voit affecter (après conversion éventuelle) la valeur retournée par l'expression de droite.

On peut donc écrire :

```
alpha = beta = 0;
```

qui est équivalent à : `alpha = ( beta = 0 );` car l'opérateur `=` est associatif de droite à gauche (c.f. le tableau des priorités et associativités page 35).

La valeur `0` est affectée à `beta` puis la valeur retournée par cette affectation est mise dans `alpha`.

3.4.3 Affectation combinée

Il existe en C un ensemble d'opérateurs d'affectations.

Syntaxe :

```
lvalue  op=  expression
```

où `op` est un des opérateurs suivants :

`*  /  %  +  -  <<  >>  &  ^  |`

`A op= B` est équivalent à `A = A op B`.

Exemple :

```
x += 3;   /* equivalent a :    x = x + 3   */
```

Un seul accès à la variable à gauche de l'opérateur est effectué.

[1] lvalue comme left value

3.5 Incrémentation/décrémentation

On doit souvent incrémenter ou décrémenter une variable. Le langage C offre 2 opérateurs (unaires) pour effectuer ces opérations :

- `++` : incrémentation de 1
- `--` : décrémentation de 1

3.5.1 Pré-incrémentation/pré-décrémentation

Les opérateurs d'incrémentation/décrémentation sont placés devant la variable. L'incrémentation/décrémentation est effectuée puis l'utilisation de la variable est faite.

Exemple : l'instruction : `v1 = ++v2;` est équivalente à :

```
v2 = v2 + 1;
v1 = v2;
```

La valeur de l'expression `++v2` est la valeur de `v2` **après** incrémentation.

3.5.2 Post-incrémentation/post-décrémentation

Les opérateurs d'incrémentation/décrémentation sont placés après la variable. L'utilisation de la variable est effectuée avant l'incrémentation/décrémentation.

Exemple : l'instruction : `v1 = v2++;` est équivalente à :

```
v1 = v2;
v2 = v2 + 1;
```

La valeur de l'expression `v2++` est la valeur de `v2` **avant** incrémentation.

3.6 L'opérateur de taille

L'opérateur `sizeof` est utilisé pour déterminer la taille (en octets) d'une variable ou d'un type.

Syntaxe :

```
sizeof(variable)     ou     sizeof variable
sizeof(type)
```

Exemple :

```
printf("taille d'une variable entiere %d\n",sizeof i);
printf("Taille d'un type entier %d",sizeof(int));
```

- L'opérateur `sizeof` appliqué à une chaîne de caractères retourne sa taille y compris le caractère `NULL` de fin de chaîne.

- L'opérateur `sizeof` appliqué à un tableau retourne la taille en octets pour stocker celui ci.

```
#include <stdio.h>

main(void) {
  int tab[10];

  printf("taille d'un tableau de 10 entiers : %d\n", sizeof(tab));
  printf("Taille de la chaine HELLO : %d", sizeof("HELLO"));
}
/*-- resultat de l'execution --------------------------
taille d'un tableau de 10 entiers : 40
Taille de la chaine HELLO : 6
-----------------------------------------------------*/
```

3.7 L'opérateur de séquence

L'opérateur , (virgule) permet d'évaluer les différentes expressions dans l'ordre. La valeur retournée est la valeur de la dernière expression évaluée.

Exemple :

```
a = ( b=10 , b+20 );
/* equivalent :   b = 10;   a = b + 20;  */

temp = a , a = b , b = temp;
```

3.8 Conversions

3.8.1 Conversions implicites

Le langage C possède ses règles de conversion de type de données à l'intérieur d'une expression. Le but de ces conversions est d'obtenir une meilleure précision du résultat.

Un certain nombre de conversions de type sont implicitement appliquées :

1. Les opérandes de type `char`, `unsigned char` et `short` sont convertis en `int`
2. Si un opérande est un `long double`, l'autre est converti en `long double`
3. Si un opérande est un `double`, l'autre est converti en `double`
4. Si un opérande est un `float`, l'autre est converti en `float`
5. Des promotions sont effectuées sur les 2 opérandes d'après les règles suivantes :
   (a) Si un opérande est un `unsigned long`, l'autre est converti en `unsigned long`

(b) Si un opérande est un `long` et l'autre est un `unsigned`, les 2 opérandes sont convertis en `unsigned long`

(c) Si un des opérandes est un `long`, l'autre est converti en `long`

(d) Si un des opérandes est un `unsigned`, l'autre est converti en `unsigned`

(e) Sinon le résultat est un `int`.

Le résultat sera du type des opérandes.

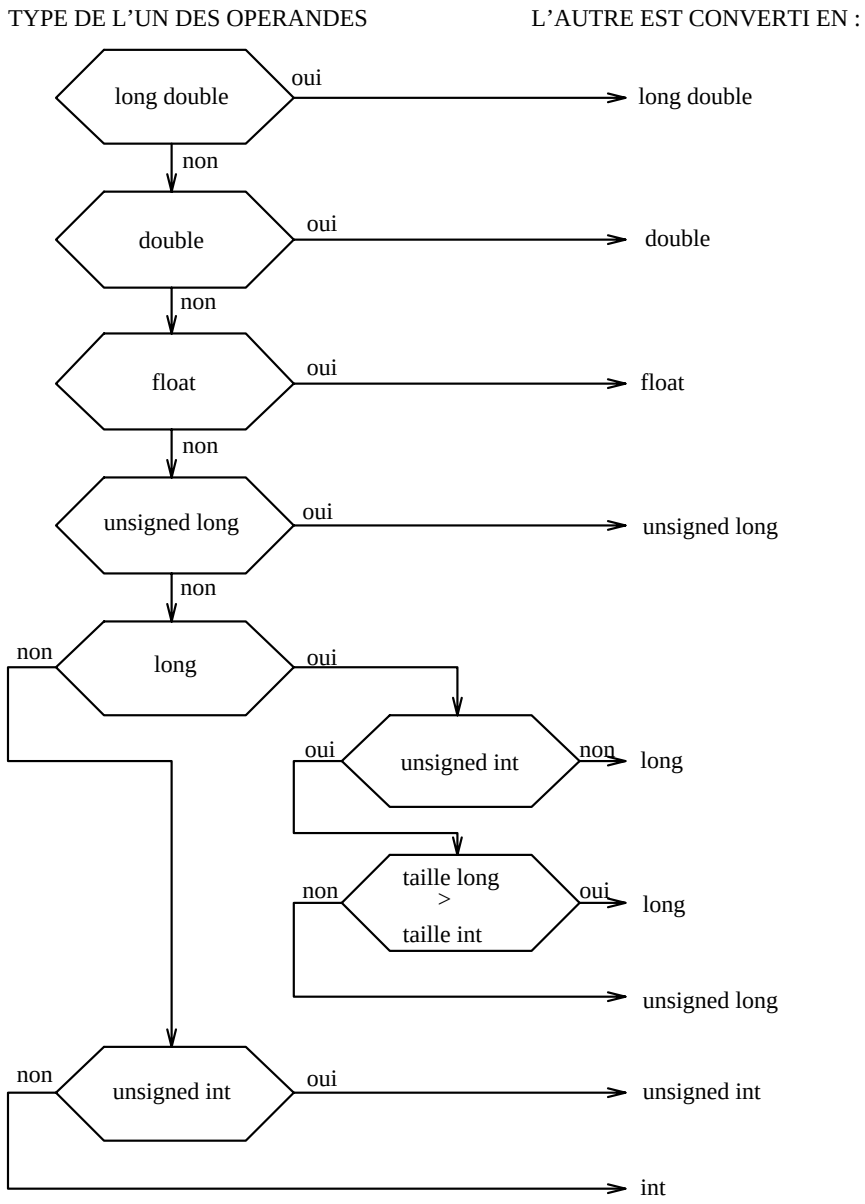


Figure 3.4: Conversions implicites

3.8.2 Conversions explicites (casting)

On peut explicitement demander une conversion dans un type désiré. Cette opération s'appelle casting ou transtypage.

Une expression précédée par un nom de type entre ( ) (parenthèses) provoque la conversion de celle ci dans le type désiré.

Exemple :

```
int   i;
float x = 12.3;
i = (int) x;   /* conversion explicite de x en entier */
```

Le casting est souvent utilisé pour s'assurer que les paramètres d'une fonction sont de type correct.

Il est recommandé de faire un large usage de cet opérateur, même quand une règle de conversion implicite s'applique.

3.9 Opérateurs relationnels

Le résultat de la comparaison entre 2 expressions vaut :

- **0** si le résultat de la comparaison est **faux**
- **1** si le résultat de la comparaison est **vrai**.

Opérateur	signification
==	test si égalité
!=	test si différent
<	test si inférieur
<=	test si inférieur ou égal
>	test si supérieur
>=	test si supérieur ou égal

Figure 3.5: Opérateurs relationnels

Exemples :

- `rep = (c > 5);`

  `rep` est égal à 1 si `c` est supérieur à 5, sinon `rep` est égal à 0.

- `if ( a == b )`

  teste si `a` est égal à `b`

3.10 Les expressions booléennes

3.10.1 Généralités

Il n'existe pas en C de type booléen. La convention suivante est utilisée :

- Une expression est vraie si elle est non nulle
- Une expression est fausse si elle est égale à zéro.

Exemples :

`5`	est une expression toujours vraie
`i = 5`	est une expression toujours vraie (affectation)
`i == 5`	est une expression vraie si `i` est égal à `5`

3.10.2 Opérateurs booléens

- `&&` Réalise le "ET booléen" entre 2 expressions. Retourne `1` (vrai) si les 2 expressions valent `1`, `0` sinon.
- `||` Réalise le "OU booléen" entre 2 expressions. Retourne `1` (vrai) si l'une ou l'autre (ou les deux), des 2 expressions valent `1`, `0` sinon.

op1	op2	op1 && op2	op1 || op2
0	0	0	0
0	non nul	0	1
non nul	0	0	1
non nul	non nul	1	1

Figure 3.6: Table de vérité des opérateurs booléens

- `!` Réalise la "négation booléenne" de son unique opérande (opérateur unaire). Retourne `1` si la valeur de son opérande est `0` (faux), `0` sinon.

op1	! op1
0	1
non nul	0

Figure 3.7: Table de vérité de l'opérateur `!`

L'évaluation des expressions booléennes s'effectue de gauche à droite (sauf pour l'opérateur `!`) et elle est stoppée dès que le résultat de l'expression devient définitif (short evaluation).

Exemples :

- `a >= 5 && a <= 10`
  - retourne `1` si `a` est compris entre `5` et `10`.
  - du fait de la priorité plus forte des opérateurs `>=` et `<=` par rapport à `&&` (c.f. tableau des priorités page 35), cette expression est équivalente à :

    ```
    (a >= 5) && (a <= 10)
    ```

- `a > 5 || b == 0`
  - retourne `1` si `a` est supérieur à `5` ou si `b` est égal à `0`.
  - l'expression `b == 0` ne sera pas évaluée si `a` est supérieur à `5` (évaluation courte).

3.11 L'opérateur conditionnel

Il permet d'écrire des expressions dont le résultat est fonction de certaines conditions.

Syntaxe :

```
( expression ) ? expression1 : expression2 ;
```

Si `expression` est vrai, le résultat est `expression1` sinon le résultat est `expression2`. `expression1` et `expression2` doivent être du même type.

Exemple :

```
n = ( a < 0 ) ? -a : a ;
```

`n` contient la valeur absolue de `a`.

Attention : Ne pas mettre un ; avant le :

3.12 Analyse lexicale

L'analyseur lexical du compilateur C extrait les unités syntaxiques de l'instruction de gauche à droite en prenant à chaque fois l'unité syntaxique la plus longue.
Ainsi, l'expression `a+++b` est évaluée en `a++ + b` et non en `a + ++b`.

C'est une très mauvaise habitude d'écrire une expression sans mettre d'espace entre les différentes éléments syntaxiques de l'expression.

3.13 Priorité et associativité des opérateurs

La priorité et l'associativité des opérateurs du langage C sont donnés par la figure 3.8 (page 35). Les opérateurs de priorité 1 sont les opérateurs de plus forte priorité.

Priorité	Opérateurs	Symbole	associativité
1	fonction	`()`	$\Longrightarrow$
	tableau	`[]`	$\Longrightarrow$
	champ de structure	`->` `.`	$\Longrightarrow$
2	négation booléenne	`!`	$\Longleftarrow$
	négation bit à bit	`~`	$\Longleftarrow$
	incrémentation	`++`	$\Longleftarrow$
	décrémentation	`--`	$\Longleftarrow$
	`-` unaire	`-`	$\Longleftarrow$
	`+` unaire	`+`	$\Longleftarrow$
	indirection	`*`	$\Longleftarrow$
	adresse	`&`	$\Longleftarrow$
	taille	`sizeof`	$\Longleftarrow$
3	casting	`(type)`	$\Longleftarrow$
4	multiplication	`*`	$\Longrightarrow$
	division	`/`	$\Longrightarrow$
	modulo	`%`	$\Longrightarrow$
5	addition	`+`	$\Longrightarrow$
	soustraction	`-`	$\Longrightarrow$
6	décalages	`<<` `>>`	$\Longrightarrow$
7	relations logiques	`<` `<=` `>` `>=`	$\Longrightarrow$
8	égalité	`==` `!=`	$\Longrightarrow$
9	ET bit à bit	`&`	$\Longrightarrow$
10	OU exclusif bit à bit	`^`	$\Longrightarrow$
11	OU bit à bit	`|`	$\Longrightarrow$
12	ET booléen	`&&`	$\Longrightarrow$
13	OU booléen	`||`	$\Longrightarrow$
14	opérateur conditionnel	`?:`	$\Longleftarrow$
15	affectations	`=` `*=` `/=` `%=`	$\Longleftarrow$
		`+=` `-=` `<<=` `>>=`	$\Longleftarrow$
		`&=` `^=` `|=`	$\Longleftarrow$
16	séquence	`,`	$\Longrightarrow$

Figure 3.8: Priorité et associativité des opérateurs

- Les expressions sont évaluées en fonction de **l'ordre de priorité** des opérateurs.

  Ainsi, l'opérateur `*` ayant une priorité supérieure à l'opérateur `+`, l'expression

  ```
  x + y * z
  ```

  est évaluée comme

  ```
  x + (y * z)
  ```

- Les **parenthèses** permettent d'outrepasser ces règles de priorité, en forçant le calcul de l'expression qu'elles contiennent.

  Elles permettent aussi une meilleure lisibilité des expressions.

- En cas d'opérateurs de même priorité, c'est **l'associativité** de l'opérateur qui détermine le sens d'évaluation de l'expression.

  L'opérateur `+` étant associatif de gauche à droite, l'expression

  ```
  x + y + z
  ```

  est évaluée comme

  ```
  (x + y) + z
  ```

- **Attention :**

  Le langage C ne précise pas dans quel ordre sont évalués les opérandes d'un opérateur.

  Dans l'expression

  ```
  tab[i] = ++i
  ```

  l'indice de `tab` peut être `i` ou `i+1`.

  De même, dans l'instruction

  ```
  tab[i] = f1();
  ```

  où la fonction `f1` modifie la valeur de `i`, l'index de `tab` pourra être celui de `i` avant ou après l'appel de la fonction.

Chapitre 4

Les structures de contrôle

4.1 L'instruction `if`

- **Syntaxe :**

```
if ( expression )
   instruction1
[else
   instruction2]
```

- **Description :** La valeur de l'expression est évaluée et, si elle est non nulle, `instruction1` est exécutée sinon c'est `instruction2` qui est exécutée (si elle existe). `instruction1` et `instruction2` peuvent être des instructions simples ou des blocs. La clause `else` peut être omise et se rapporte toujours au dernier `if` visible.

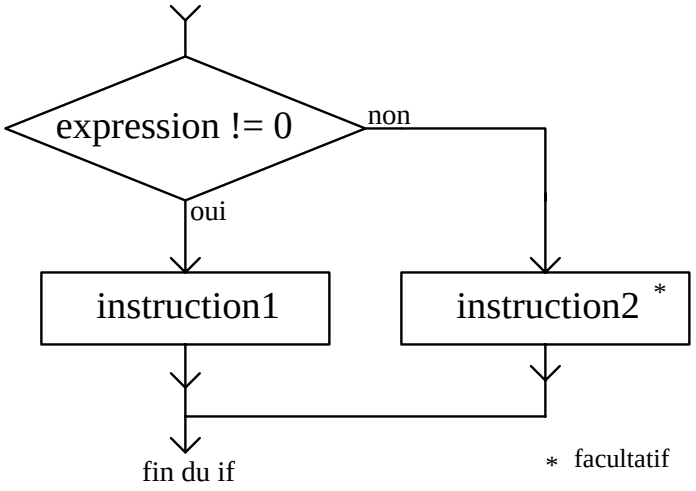


Figure 4.1: Diagramme syntaxique du `if`

- **Exemples :**
  -
```
if (i < 10)
    i++;
```

```
- if (delta > 0)
     printf("2 racines reelles\n";
  else
    if (delta == 0)
       printf("racine double\n");
    else
       printf("pas de racines reelles\n");
- if (a)        /* equivalent :  if ( a != 0 )  */
     i++;
- if (an % 4 == 0 && an % 100 != 0 || an % 400 == 0)
      printf("annee bissextile\n");
```

4.2 L'instruction `while`

Cette instruction permet de répéter une instruction (ou un bloc) tant qu'une condition est vraie.

- **Syntaxe :**

```
while ( expression )
   instruction ou bloc d'instructions
```

- **Description :**

`instruction` est exécutée de façon répétitive aussi longtemps que le résultat de `expression` est non nul. `expression` est évaluée avant chaque exécution de `instruction`.

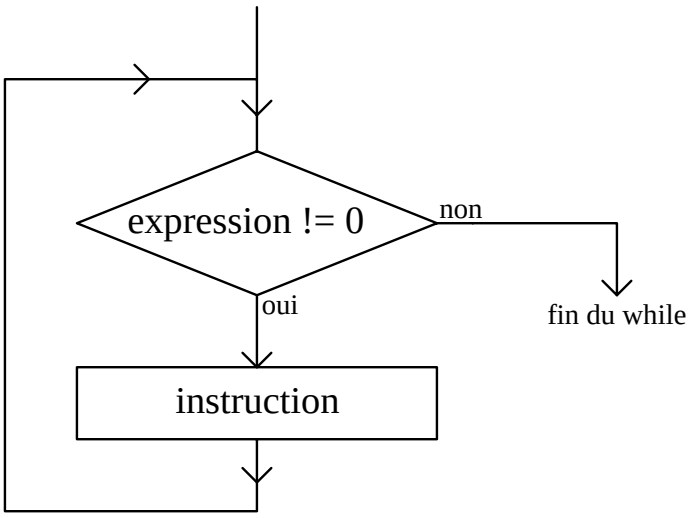


Figure 4.2: Diagramme syntaxique du `while`

- **Exemple :**

```
int i = 0 ;
/* affiche les nombres de 0 - 9 */
while (i != 10) {
   printf("%d ", i);
   i++;
}
```

4.3 L'instruction `do ... while`

- **Syntaxe :**

```
do
    instruction ou bloc d'instructions
while ( expression != 0 );
```

- **Description :**

Cette instruction est similaire à la précédente, le test a lieu après chaque exécution de l'instruction, de fait l'instruction est au moins exécutée une fois.

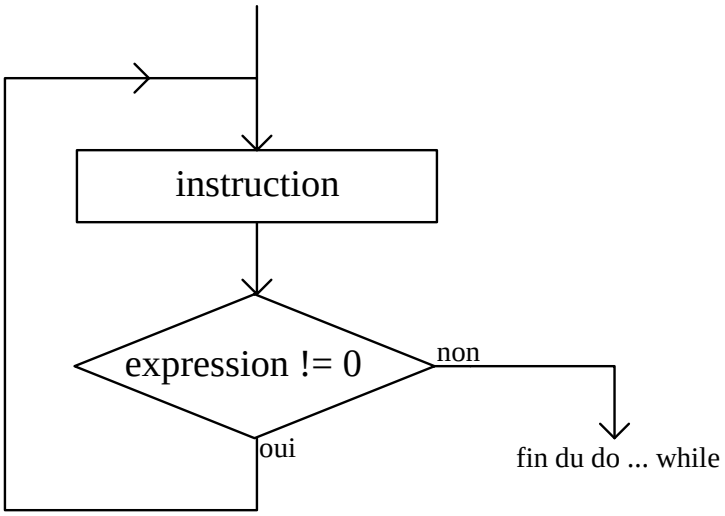


Figure 4.3: Diagramme syntaxique du `do ... while`

- **Exemples :**

  - ```
    int i = 1 , somme = 0;
    /* somme des nombres de 1 a 10 */
    do {
      somme += i;
      ++i;
    } while (i <= 10);
    ```
    ```

```
char c;
/* test de reponse */
do {
  printf("Voulez-vous continuer (o/n) ? ");
  c = getchar();
} while (c != 'o' && c != 'n');
```

4.4 L'instruction `for`

- **Syntaxe :**

```
for ( [expression1] ; [expression2] ; [expression3] )
   instruction ou bloc d'instructions
```

- **Description :**

  `instruction` est répétée tant que la valeur de `expression2` est non nulle.

  Avant la première itération, `expression1` est évaluée; En général elle sert à initialiser les variables de la boucle.

  Après chaque itération de la boucle, `expression3` est évaluée. En général elle sert à incrémenter le compteur de la boucle.

  La boucle `for` est équivalente à la structure suivante :

```
expression1;
while ( expression2 ) {
   instruction
   expression3;
}
```

- **Remarques :**

  - Toutes les expressions sont facultatives. Si `expression2` n'existe pas, elle sera supposée vraie (valeur 1).
  - L'opérateur `,` (virgule c.f. page 30) peut être utilisé dans `expression1` et `expression3`.

- **Exemples :**

  - ```
    for (i=0; i<10; i++)
       somme += tab[i];
    ```
  - ```
    for( ; ; )
       ;          /* boucle infinie ne faisant rien ! */
    ```
  - ```
    for( i = 0 , j = 0 ; i < 10 ; i++ , j++ )
       ...
    ```
    ```

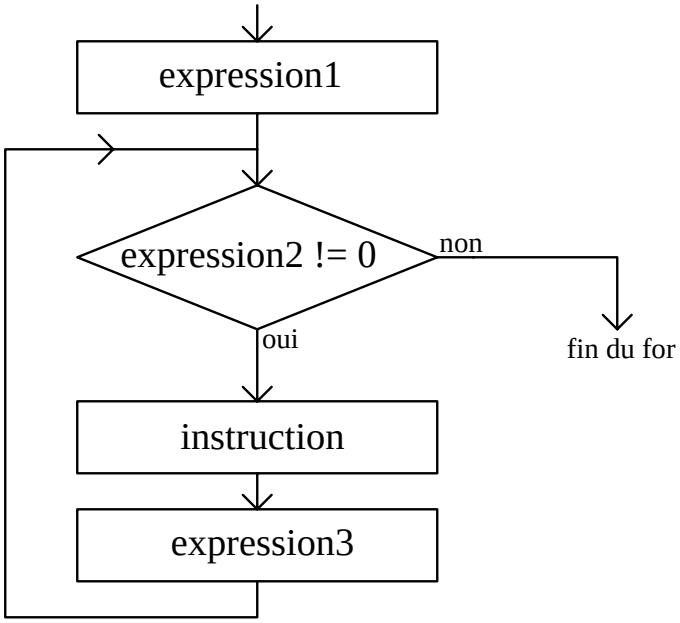


Figure 4.4: Diagramme syntaxique du `for`

4.5 L'instruction `switch`

L'instruction `switch` permet un choix multiple en fonction de l'évaluation d'une expression.

- **Syntaxe :**

```
switch ( expression ) {
     case e1 : instruction1 ...
     case e2 : instruction2 ...
     ...
     case e3 : instruction3 ...
     default : instruction_default ...
}
```

- **Description :**

  L'évaluation de `expression` doit donner pour résultat une valeur de type `int`.

  `e1, e2, e3 ...` sont des expressions constantes qui doivent être un entier unique de type `int` ou `char`.

  La valeur de `expression` est recherchée successivement parmi les valeurs des différentes expressions constantes `e1, e2, e3`. En cas d'égalité les instructions (facultatives) correspondantes sont exécutées jusqu'à une instruction `break` (cf. 4.6 page 42) ou jusqu'à la fin du bloc du `switch` (et ceci indépendamment des autres conditions `case`).

  S'il n'y a pas de valeur correspondante, on exécute les instructions du cas `default` (s'il existe).

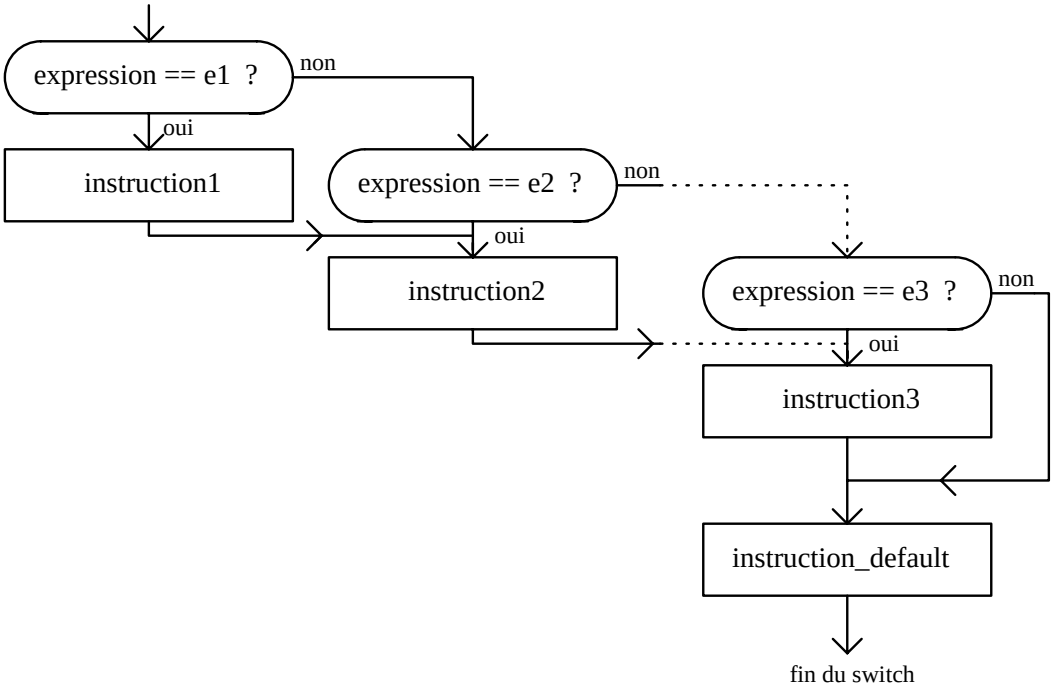


Figure 4.5: Diagramme syntaxique du `switch`

- **Exemple :**

```
switch (car) {
     case 'a' :
     case 'A' :
     case 'e' :
     case 'E' : v++;
                break;
     case ' ' : espace++;
                break;
     default  : c++;
}
```

4.6 L'instruction `break`

L'instruction `break` provoque le passage à l'instruction qui suit immédiatement le corps de la boucle `while`, `do-while`, `for` ou `switch`.

4.7 L'instruction `continue`

L'instruction `continue` fait passer à l'itération suivante les instructions `while`, `do-while` ou `for`. Elle est équivalente à un "`goto`" à la fin du bloc.

Exemple :

```
for (i = -10; i <= 10; i++) {
  if (i == 0)
     continue;
  tab[i] = 1 / i;
}
```

Pour la boucle `for`, l'instruction `continue` fait passer à l'évaluation de `expr3`.

Chapitre 5

Les pointeurs

La spécificité du langage C vient de son traitement des pointeurs. Leur utilisation est donc très importante.

Un pointeur est une variable qui contient l'adresse d'une autre variable de n'importe quel type. Sa taille dépend de l'implémentation et est, en général, de la taille d'un `int`.

5.1 Définition d'une variable pointeur

La syntaxe générale d'une définition ou déclaration de variable pointeur est de la forme :

```
classe modificateur type *identificateur ;
```

La classe, le modificateur et le type sont ceux de la variable pointée.

Exemple :

```
char *pt1;
```

`pt1` est une variable pointeur sur un `char`. Elle contiendra donc l'adresse d'une zone mémoire capable de contenir elle même un objet de type `char`.

Un pointeur sur un type `void` pointe sur une zone sans type.

Exemple :

```
void *pt2;
```

`pt2` est une simple adresse mémoire, indépendante du type de l'objet pointé, et qui peut être uniquement affecté à un autre pointeur.

- La définition d'un pointeur n'alloue en mémoire que la place nécessaire pour stocker ce pointeur.
- un pointeur (comme toute autre variable) doit être initialisé avant d'être utilisé.

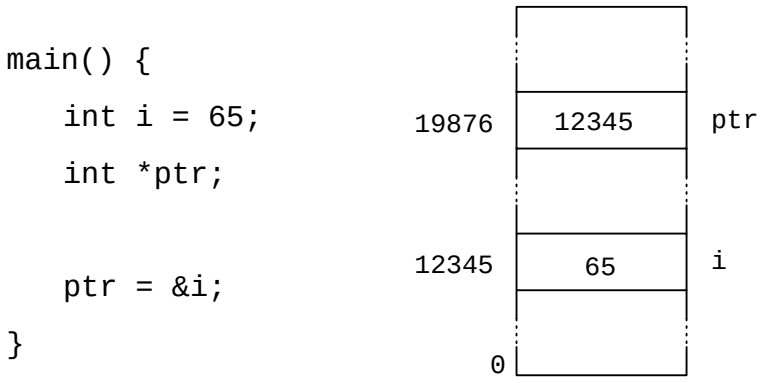


Figure 5.1: Représentation en mémoire d'un pointeur

5.2 Opérateur d'adresse

L'opérateur unaire `&` fournit l'adresse en mémoire d'une variable C. Il permet donc d'initialiser la valeur d'un pointeur (c.f. figure 5.1).

5.3 Opérateur d'indirection

L'opérateur unaire d'indirection `*` (étoile) permet d'obtenir la valeur d'un élément dont l'adresse est contenue dans le pointeur.

Exemple :

```
int   i = 65 , *ptr;

ptr = &i;         /* ptr contient l'adresse de i   */
*ptr = *ptr + 1;  /* equivalent a :     i = i + 1; */
(*ptr)++;         /* equivalent a :     i++;       */
```

Il n'est pas possible d'effectuer une indirection sur un pointeur de type `void`.

Exemple :

```
void  *ptr1;
int   *ptr2;

*ptr1 = 10;            /* INCORRECT   */
ptr2  = (int *) ptr1;  /* CORRECT     */
```

5.4 La constante `NULL`

Elle définit la valeur d'un pointeur ne pointant sur rien. Sa valeur est l'entier 0. Elle est utilisée pour des raisons de lisibilité et de sécurité.

Exemple :

```
int *ptr = NULL;
```

définit un pointeur sur un entier et l'initialise avec la constante `NULL`. On peut l'utiliser de cette façon :

```
if (ptr == NULL)
   exit(0);
```

5.5 Arithmétique des pointeurs

5.5.1 Incrémentation/décrémentation de pointeur

L'incrémentation (ou la décrémentation) de pointeur exprime le fait que l'on veuille atteindre l'élément pointé suivant (ou précédent). Le résultat est correct que si l'objet pointé est situé dans un même tableau.

Ainsi avec la définition suivante : `int *ptr;` les instructions `ptr++;` ou `ptr = ptr + 1;` sont équivalentes à incrémenter la valeur du pointeur de la longueur du type de l'objet pointé : `ptr = ptr + sizeof( int );`

Ces opérations sont impossibles avec des pointeurs sur un type `void`.

5.5.2 Addition/soustraction de 2 pointeurs

- l'addition de 2 pointeurs n'a pas de sens et est donc sans signification.
- la soustraction de 2 pointeurs (de même type) situés dans un **même tableau**, retourne le nombre d'éléments qui les séparent.

Exemple :

```
int tab[20] , *pt1 = &tab[2] , *pt2 = &tab[6];

printf("%d\n", pt2 - pt1);  /* affiche :  4  */
```

5.5.3 Comparaison de pointeurs

Un pointeur ne pourra être comparé qu'à un pointeur sur le même type (ou à la constante `NULL`). Le résultat de la comparaison sera significatif et portable que si les pointeurs pointent sur le même objet.

5.5.4 Affectation de pointeurs

On ne doit affecter à un pointeur qu'une valeur de pointeur sur un objet de même type. Dans le cas contraire, de telles affectations risquent de ne pas être portable (contrainte d'alignement, représentation des données en mémoire ...). On utilisera une conversion explicite (casting cf. 3.8.2 page 32) dans ce cas.

Exemple :

La fonction d'allocation dynamique `malloc` retourne un pointeur sur un `void` : si l'on veut faire une allocation dynamique d'un entier, on doit écrire (pour être portable) :

```
int *t;   /* t est un pointeur sur un entier */

t = (int *) malloc( sizeof(int));
```

`t` contient l'adresse de début d'une zone de mémoire pouvant contenir un entier.

5.6 `const` et les pointeurs

On peut, à l'aide du spécificateur `const`, s'assurer que le pointeur reste constant :

```
char * const ptr = tab;
```

`ptr` est un pointeur constant sur un caractère et initialisé avec l'adresse `tab`. On ne peut donc pas modifier la valeur de `ptr`.

Par contre la définition suivante :

```
const char *ptr ;
```

définit `ptr` comme un pointeur sur une constante caractère. On peut donc modifier la valeur de `ptr` mais pas la valeur pointée par ce dernier.

La dernière forme, résultante des 2 formes précédentes, déclare un pointeur constant sur une constante caractère :

```
const char * const ptr = tab;
```

on ne peut modifier ni le pointeur ni la valeur pointée.

5.7 Pointeurs et tableaux

- Le nom seul d'un tableau est un **pointeur constant** sur le premier élément de celui-ci.

  Avec les définitions ci dessous :

  ```
  char *ptab;      /* ptab est un pointeur sur un char */
  char tab[10];    /* tableau de 10 char               */
  ```

  on peut écrire :

  ```
  ptab = &tab[0];    qui est équivalent à :    ptab = tab;
  tab[0] = 'a';      qui est équivalent à :    *ptab = 'a';
  tab[3] = 'd';      qui est équivalent à :    *(ptab+3) = 'd';
  ```

  ainsi on peut dire que :

  ```
  identif[expression]
  ```

  est équivalent à :

```
*(identif + expression )
```

L'incrémentation d'un pointeur fait que `expression` est multiplié par la taille du type pointé (ici un `int`) avant d'être ajouté à `identif`.

- L'affectation globale entre tableaux n'est pas possible. Ainsi après la définition suivante :

  ```
  int tab1[80],tab2[80];
  ```

  l'instruction `tab1 = tab2;` n'est pas possible car le nom seul d'un tableau est un pointeur constant sur le premier élément de celui-ci. Cette écriture n'a pas plus de sens que l'expression : `2 = 3`

- En C, une chaîne de caractères est considérée comme un tableau de caractères, dont le dernier élément est le caractère `'\0'` (caractère de code 0).

- Pour travailler sur un tableau on a donc le choix entre travailler avec un tableau en utilisant un indice ou travailler avec un pointeur.

Exemple :

```
/* copie de chaine, version tableau */
void copie(char srce[] , char dest[]) {
  register int i;
  for( i=0 ; (dest[i] = srce[i]) != 0 ; i++)
     ;
}


/* copie de chaine, version pointeur */
void copie(char *srce , char *dest) {
  while ( (*dest++ = *srce++) != 0 )
     ;
}
```

5.8 Les tableaux et les chaînes

L'initialisation d'un tableau de caractères peut se faire de la manière suivante :

```
char msg[] = "Hello";
```

Cette écriture est l'abréviation de :

```
char msg[] = {'H','e','l','l','o','\0'};
```

La dimension du tableau est ici facultative, car elle peut être calculée par le compilateur. `msg` est un tableau initialisé, on peut donc modifier son contenu.

ATTENTION : en dehors de la déclaration, il n'est pas possible d'affecter une chaîne de caractères à un tableau. Il faut passer par une fonction de copie de chaînes de caractères (`strcpy` par exemple).

Par contre dans la définition suivante :

```
char *adieu = "Au revoir";
```

`adieu` est un pointeur qui contient l'adresse du premier caractère de la constante chaîne de caractères `"Au revoir"`.

D'autre part :

```
char ligne[81];

ligne =  "Bonjour";   /* INTERDIT !!! */
```

`ligne` est le nom d'un tableau, c'est un pointeur **constant** et ce n'est donc pas une "lvalue". `"Bonjour"` est une **constante** chaîne de caractères et est considérée comme une constante de type tableau de caractères.

Il est aussi abusif d'écrire `ligne = "Bonjour";` que d'écrire `2 = 3;` ou `2 = a;` !!!

Il faut employer `strcpy(ligne,"Bonjour");` (ou la fonction `copie` de l'exemple précédent).

5.9 L'opérateur d'adresse : explication complémentaire

`&operande` donne l'adresse de `operande`. `operande` doit être obligatoirement une *valeur gauche*.

Ainsi

- `&(i+1)` provoque une erreur à la compilation puisque `i+1` n'est pas une *lvalue*.
- Quelles sont les valeurs affichées par ce programme ?

```
#include <stdio.h>

main(void) {
  char ligne[80];

  /*              %p affiche la valeur du pointeur en hexadecimal*/
```

```
  printf("ligne = %p    &ligne = %p\n", ligne, &ligne);
  return 0;
}
```

Réponse : les mêmes.

`&ligne` donne l'adresse du tableau `ligne`, c'est à dire la même valeur que `ligne`. Le compilateur, dans ce cas, ignore purement et simplement l'opérateur `&`. [1]

`ligne` étant une **constante** pointeur (donc n'est pas une *valeur gauche*), l'expression `&ligne` ne devrait pas être autorisée...

Parfois des fonctions demandent des adresses de chaînes de caractères comme arguments :

```
#include <stdio.h>
#include <string.h>   /* strcmp() */

int compare(char **str1, char **str2){
  return strcmp(*str1, *str2);
}

main(void) {
  char ligne[80], *ptr1 = "DCL";
  char *ptr2;

  printf("Element : "); scanf("%s", ligne);
  printf("resultat 1 : %d\n", compare( (char **) &ligne, &ptr1) );

  ptr2=ligne;
  printf("resultat 2 : %d\n", compare( &ptr2, &ptr1) );

  return 0;
}
/*-- resultat de l'execution -------------------------
Element : DCL
resultat 1 : -22
resultat 2 : 0
------------------------------------------------------*/
```

Le premier appel à la fonction `compare` donne un résultat faux, malgré la conversion explicite `(char **)`[2].

[1] initiative malheureuse de la plupart des compilateurs ???

[2] de toute manière une conversion explicite n'a jamais changé une valeur !!! elle est uniquement à usage de précision syntaxique pour le compilateur.

Pour obtenir un résultat juste, il faut opérer par l'intermédiaire d'une autre variable comme le montre le deuxième appel à la fonction `compare`.

5.10 Tableau de pointeurs

Un tableau de chaîne de caractères pourrait être défini comme un tableau à 2 dimensions :

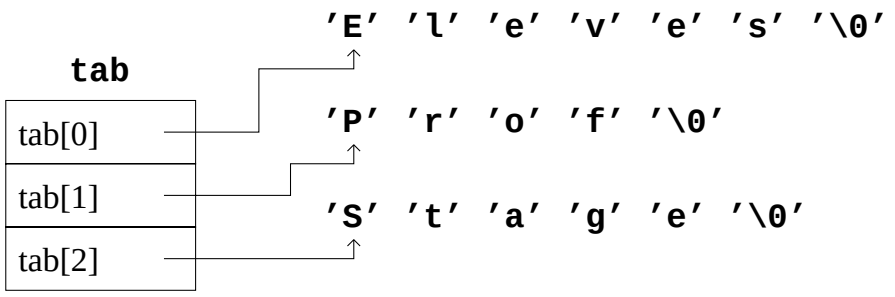
`char tab[10][81];`
Cette définition réserve en mémoire 810 octets,

On préfère définir un tableau de pointeurs sur des chaînes :

`char *tab[10];`
Ce tableau de 10 pointeurs sur des chaînes permet à ces dernières d'être de longueur variable et d'occuper en mémoire que la place nécessaire.

Exemple :

```
char *tab[ ] = { "Eleves","Prof","Stage"};
```



`tab` : est équivalent à : `&tab[0]` : c'est l'adresse du premier élément du tableau.

`*tab` : est équivalent à : `tab[0]` : c'est le premier élément du tableau, c'est à dire l'adresse du premier caractère de la chaîne `"Eleves"`.

`tab[1]` : est l'adresse sur le premier caractère de la chaîne `"Prof"`.

`*(tab+1)` : est équivalent à : `tab[1]`

`*tab[1]` : est le caractère pointé par `tab[1]` : c'est le caractère `'P'` de la chaîne `"Prof"`.

`**tab` : est équivalent à : `*tab[0]` : c'est le caractère `'E'` de la chaîne `"Eleves"`.

`**(tab+1)` : est équivalent à : `*tab[1]` : c'est le caractère `'P'` de la chaîne `"Prof"`.

`*(tab[2]+1)` : est équivalent à : `tab[2][1]` : c'est le caractère `'t'` de la chaîne `"Stages"`.

5.11 Ce programme affiche CCCCCCCC

```
#include <stdio.h>

main() {
   char *str = "ENAC";

   putchar( str[3] );          /* 1 */
   putchar( *(str + 3) );      /* 2 */
   putchar( *(3 + str) );      /* 3 */
   putchar( 3[str] );          /* 4 */

   putchar( "ENAC"[3] );       /* 5 */
   putchar( *("ENAC" + 3) );   /* 6 */
   putchar( *(3 + "ENAC") );   /* 7 */
   putchar( 3["ENAC"] );       /* 8 */
}
/*-- resultat de l'execution -------------------
CCCCCCCC
-------------------------------------------------*/
```

- **Explications des instructions 1, 2, 3 et 4 :** La notation "crochet" en langage C, n'est que du **"sucre syntaxique"**. Ainsi l'expression `str[3]` est équivalente à `*(str + 3)`. L'addition étant commutative, les instructions `/* 2 */` et `/* 3 */` du programme ci dessus sont elles aussi équivalentes, et donc aussi équivalente à `3[str]`.

- **Explications des instructions 5, 6, 7 et 8 :** Une constante chaîne de caractères est en réalité un pointeur constant sur le premier caractère de cette chaîne. Ainsi l'expression `"ENAC"[3]` nous retourne le caractère `C` de la chaîne `"ENAC"`.

  Cette constante chaîne étant un pointeur (constant), les expressions `/* 6 */`, `/* 7 */` et `/* 8 */` sont donc équivalentes (c.f. le principe énoncé pour l'explication des instructions `/* 1 */` à `/* 4 */`).

Chapitre 6

Affichages et saisies

Les fonctions suivantes qui permettent d'afficher sur l'écran ou de saisir au clavier, font partie de la bibliothèque standard d'entrées/sorties.

6.1 La fonction `printf`

- **Syntaxe :**

  ```
  #include <stdio.h>

  int printf( const char *format [, arg [, arg]...]);
  ```

- **Description :**

  Elle permet l'écriture formatée sur le flux standard de sortie `stdout` (l'écran par défaut).

  La chaîne de caractères `format` peut contenir à la fois :

  1. Des caractères à afficher,
  2. Des spécifications de format.

  Il devra y avoir autant d'arguments à la fonction `printf` qu'il y a de spécifications de format.

- **Valeur retournée :** le nombre d'octets effectivement écrits ou la constante `EOF` en cas d'erreur.

- **Spécificateurs de format :** ils sont introduites par le caractère % et se terminent par le caractère de type de conversion suivant la syntaxe suivante :

  ```
  % [drapeaux] [largeur] [.precision] [modificateur] type
  ```

  - `drapeaux` :

drapeaux	Signification
`rien`	justifié à droite et complété à gauche par des espaces
`-`	justifié à gauche et complété à droite par des espaces
`+`	les résultats commencent toujours par le signe + ou -
`espace`	le signe n'est affiché que pour les valeurs négatives
`#`	forme alternative. Si le type de conversion est : c,s,d,i,u : sans effet o : un `0` sera placé devant la valeur x, X : `0x` ou `0X` sera placé devant la valeur e, E, f : le point décimal sera toujours affiché g, G : même chose que e ou E, mais sans supprimer les zéros à droite

- `largeur` : elle précise la nombre de caractères `n` qui seront affichés.

  Si la valeur à afficher dépasse la taille de la fenêtre ainsi définie, C utilise quand même la place nécessaire.

largeur	Effet sur l'affichage
`n`	affiche n caractères, complété éventuellement par des espaces
`0n`	affiche n caractères, complété éventuellement à gauche par des `0`
`*`	l'argument suivant de la liste fournit la largeur

- `precision` : elle précise pour

  * un entier, le nombre de chiffres à afficher
  * un réel, le nombre de chiffres de la partie décimale à afficher (avec arrondi)
  * les chaînes, le nombre maximum de caractères à afficher.

.precision	Effet sur l'affichage
`rien`	précision par défaut d,i,o,u,x : 1 chiffre e, E, f : 6 chiffres pour la partie décimale.
`.0`	d,i,o,u,x : précision par défaut e, E, f : pas de point décimal
`.n`	n caractères au plus
`*`	l'argument suivant de la liste contient la précision

- `modificateur` : Il précise comment sera interprété l'argument.

Modificateur	interprétation comme
`h`	un entier de type short (d,i,o,u,x,X)
`l`	un entier de type long (d,i,o,u,x,X)
`L`	un réel de type long double (e,E,f,g,G)

- `type` : type de conversion de l'argument.

Type	Format de la sortie
d ou i	entier décimal signé
o	entier octal non signé
u	entier décimal non signé
x	entier hexadécimal non signé
X	entier hexadécimal non signé en majuscules
f	réel de la forme [-]dddd.ddd
e	réel de la forme [-]d.ddd e [+/-]ddd
E	comme e mais l'exposant est la lettre E
g	format e ou f suivant la précision
G	comme g mais l'exposant est la lettre E
c	caractère
s	affiche les caractères jusqu'au caractère nul '\0' ou jusqu'à ce que la précision soit atteinte
p	pointeur

- **Exemple :**

```
#include <stdio.h>

main(void) {
  int nbre = 5;
  char *chaine = "Le langage C";
  long prix = 12.0L;
  long double result = prix * nbre;

  printf("Bonjour\n");
  printf("Nombre %d  prix %ld Total %9ld\n",nbre, prix, prix * nbre);
  printf("%s est facile\n", chaine);
  printf("%8.2Lf \n", result);
  printf("%*.*Lf \n", 8, 2, result);  /* equivalent a   %8.2Lf    */
  printf("%%\n");                     /* affichage du caractere % */
  return 0;
}
/*-- resultat de l'execution ------------------------------------
Bonjour
Nombre 5  prix 12 Total        60
Le langage C est facile
   60.00
   60.00
%
-----------------------------------------------------------------*/
```

- **Exemple d'utilisation des formats numériques :**

instruction C	résultat
`printf("|%d|\n",12345);`	`|12345|`
`printf("|%+d|\n",12345);`	`|+12345|`
`printf("|%8d|\n",12345);`	`|   12345|`
`printf("|%8.6d|\n",12345);`	`|  012345|`
`printf("|%x|\n",255);`	`|ff|`
`printf("|%X|\n",255);`	`|FF|`
`printf("|%#x|\n",255);`	`|0xff|`
`printf("|%f|\n",1.23456789012345);`	`|1.234568|`
`printf("|%10.4f|\n",1.23456789);`	`|    1.2346|`

6.2 La macro putchar

- **Syntaxe :**

```
#include <stdio.h>

int putchar(int c);
```

- **Description :** elle permet d'écrire, à la position courante, le caractère `c` sur la sortie standard.

- **Valeur retournée :**
  - En cas de succès : la valeur effectivement envoyée
  - En cas d'erreur : la constante `EOF` et positionne `errno` pour indiquer l'erreur.

- **Remarque :**

  L'appel de cette macro est bien plus rapide qu'un appel à la fonction `printf("%c",c);` dans la mesure où elle ne fait pas appel à l'analyse d'une chaîne de format.

  Cette macro est définie dans `stdio.h` comme :

```
#define putchar(x)  putc((x), stdout)
```

- **Exemple :**

```
#include <stdio.h>

main(void) {
  char msg[] = "Appuyer sur une touche pour continuer";
  int i;

  for (i=0 ; msg[i] != '\0' ; i++)
     putchar(msg[i]);
  putchar('\n');
  return 0;
}
```

6.3 La fonction `puts`

- **Syntaxe :**

```
#include <stdio.h>

int puts(const char *s);
```

- **Description :**

  Elle permet d'écrire, à la position courante, la chaîne de caractères pointée par `s` sur la sortie standard.

- **Valeur retournée :**

  - En cas de succès : une valeur non négative
  - En cas d'erreur : la constante `EOF` et positionne `errno` pour indiquer l'erreur.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  char msg = "Appuyer sur une touche pour continuer\n";

  puts(msg);
  return 0;
}
```

6.4 La fonction `scanf`

- **Syntaxe :**

```
#include <stdio.h>

int scanf( const char *format [, arg [, arg]...]);
```

- **Description :**

  La fonction `scanf` permet de faire une lecture formatée du flux standard d'entrée (le clavier par défaut).

  Elle lit les caractères en entrée, les interprète en concordance avec les spécifications de format décrites dans la chaîne `format`, et place les résultats dans les arguments `arg`.

  Pour pouvoir retourner les valeurs ainsi saisies, les `arg` doivent être obligatoirement des **pointeurs**.

- **Valeur retournée :** le nombre de valeurs convenablement introduites ou `EOF` en cas d'erreur.

- **Remarques :**

  Les informations tapées au clavier sont d'abord mémorisées dans un **tampon** avant d'être traitées par `scanf`.

  La chaîne de format ne doit comporter que des spécifications de format, tout autre caractère peut amener le programme à se comporter curieusement ...

- **Spécificateurs de format**

  Ils sont introduits par le caractère `%` et se terminent par le caractère de type de conversion suivant le format suivant :

  ```
  % [largeur] [modificateur] type
  ```

  - `largeur` : elle précise la nombre de caractères `n` qui seront lus. On peut en lire moins si l'on rencontre un séparateur (espace, tabulation, retour chariot ...) ou un caractère invalide.

  - `modificateur` : Il précise la taille de l'objet recevant la valeur.

Modificateur	l'objet recevant est
`h`	un entier de type `short int` (d,i,o,u,x)
`l`	un entier de type `long int` (d,i,o,u,x)
	un réel de type `double` (e,f)
`L`	un réel de type long double (e,f,g)

  - `type` : type de l'objet pointé par `arg`.

Type	Type de l'objet pointé
`d`	`signed int` exprimé en décimal
`o`	`signed int` exprimé en octal
`u`	`unsigned int` exprimé en décimal
`x`	`int` (`signed` ou `unsigned`) exprimé en hexadécimal
`f,e,g`	réel
`c`	suivant la `largeur` : `largeur` non spécifiée ou égale à 1 : caractère `largeur` différente de 1 : une chaîne de caractères
`s`	une chaîne de caractères
`p`	pointeur exprimé en hexadécimal

- **Exemple :**

```
#include <stdio.h>

main(void) {
  int i,j;
  double d;
  char tab[81];
```

```
  printf("entier: ");
  scanf("%d", &i);
  printf("2 entiers et 1 double: ");
  scanf("%d%d%lf", &i, &j, &d);
  printf("chaine (sans espace): ");
  scanf("%80s", tab);
  /* le caractere '\0' est automatiquement ajoute a la fin de la chaine tab*/
  return 0;
}
```

6.5 La macro `getchar`

- **Syntaxe :**

```
#include <stdio.h>

int getchar();
```

- **Description :**

  Elle permet la lecture d'un caractère sur l'entrée standard.

- **Valeur retournée :** cette macro retourne :
  - En cas de succès : la valeur du caractère lu
  - En cas d'erreur : la constante `EOF` et positionne `errno` pour indiquer l'erreur.

- **Remarques :**
  - L'appel de cette macro est bien plus rapide qu'un appel à la fonction `scanf("%c",&c);` dans la mesure où elle ne fait pas appel à l'analyse d'une chaîne de format.
  - Cette macro est définie dans `stdio.h` comme :

    ```
    #define getchar()  getc(stdin)
    ```

  - La macro `getchar` utilise le même tampon que la fonction `scanf`.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  int rep;

  do {
    printf("Voulez-vous continuer ? ");
    rep = getchar();
  } while ( rep != 'o' && rep != 'n');
  return 0;
}
```

```
/*-- resultat de l'execution --------------------------------
Voulez-vous continuer ? u
Voulez-vous continuer ? Voulez-vous continuer ? tr
Voulez-vous continuer ? Voulez-vous continuer ? Voulez-vous continuer ?
-------------------------------------------------------------*/
```

- **Remarques sur le résultat de l'exécution :**
  - le `'\n'` ainsi que les autres caractères frappés restent dans le buffer du flux d'entrée standard et le prochain `getchar()` extraira le premier d'entre eux du buffer.
  - la solution consiste à ne pas bufférriser les lectures du tampon d'entrée standard :

```
#include <stdio.h>

main(void) {
  int rep;

  system("stty raw");
  do {
    printf("Voulez-vous continuer ? ");
    rep = getchar();
  } while ( rep != 'o' && rep != 'n');
  system("stty cooked");
  return 0;
}
```

6.6 La fonction `gets`

- **Syntaxe :**

```
#include <stdio.h>

char *gets(char *s);
```

- **Description :**

  Elle permet la lecture d'une chaîne de caractères, terminée par `\n` (newline), sur l'entrée standard et place le résultat dans le tableau pointé par `s`.

- **Valeur retournée :** cette fonction retourne :
  - En cas d'erreur : le pointeur `NULL` et positionne `errno` pour indiquer l'erreur.
  - en cas de succès : le pointeur `s`.

- **Remarques** :
  - Le caractère `\n` (newline) n'est pas recopié dans la chaîne pointée par `s` et un caractère nul (`'\0'`) est placé à la fin de la chaîne.

- La fonction `scanf("%s",s);` ne permet pas de lire des chaînes de caractères contenant des espaces (l'espace étant pour `scanf` un séparateur), tandis qu'avec `gets`, seul le caractère `\n` (newline) sert de délimiteur.
- la fonction `gets` ne gère pas des débordements éventuels sur la chaîne passée en paramètre. Pour cela il vaudra mieux utiliser le fonction `fgets` sur le flux `stdin`.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  char ligne[10];

  printf("Votre texte 1 ? ");
  gets(ligne);
  printf("Votre texte 2 ? ");
  fgets(ligne, 11, stdin);
  return 0;
}
/* resultat de l'execution ----------------------------------------
Votre texte 1 ? qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
Segmentation fault (core dumped)
----------------------------------------------------------------------*/
```

6.7 Les fonctions `sscanf` et `sprintf`

- **Syntaxe :**

```
#include <stdio.h>

int sscanf(char *s, char *format, pointer ...);

int sprintf(char *s, char *format, arg ...);
```

- **Description :**
  - `sscanf` est analogue à `scanf`, sauf qu'à la place de lire l'entrée standard, elle lit la chaîne d'adresse `s`.
  - `sprintf` est analogue à `printf`, sauf qu'à la place d'écrire sur la sortie standard, elle écrit dans la chaîne d'adresse `s`

- **Exemple :**

```
#include <stdio.h>

main(void) {
```

```
  char buffer[80], *msg="Votre reponse";
  int num;

  do {
    printf( "Numero ? " );
    gets( buffer );
  } while ( sscanf(buffer, "%d", &num) != 1 && num >= 0 && num <= 255 );

  sprintf(buffer, "%s num = %d\n", msg, num);
  puts(buffer);
  return 0;
}
/*-- resultat de l'execution ---------------------------------
Numero ? abcd
Numero ? 12
Votre reponse num = 12
-------------------------------------------------------------*/
```

Chapitre 7

Les fonctions

7.1 Généralités

Les fonctions sont les briques de base d'un programme C. Elles nous permettent de découper des programmes en parties plus petites donc plus faciles à relire et à mettre au point.

Une fonction est un sous-programme qui effectue un travail bien précis, à laquelle on passe (généralement) des données et qui retourne (le plus souvent) une valeur.

Cette capacité nous permet de créer des routines indépendantes qui pourront être appelées depuis d'autres fonctions.

Quand une fonction est appelée, l'exécution du programme est transférée à la première instruction de cette fonction. L'exécution de la fonction se termine après l'instruction `return` ou après la dernière instruction du bloc de la fonction.

Quand la fonction appelée a terminé, l'exécution dans la fonction appelante, reprend à l'endroit de l'expression où l'appel avait été fait. La valeur retournée par la fonction peut être alors utilisée comme opérande dans cette expression.

L'échange de données entre fonctions est réalisé par l'intermédiaire de variables globales[1] et/ou par une liste d'arguments.

7.2 Passage des paramètres

Quand on invoque une fonction, on doit généralement lui passer des valeurs. Il y a deux points de vue pour le passage des données à la fonction, celui de la fonction appelante et celui de la fonction appelée.

- Du coté appelant, les valeurs passées sont appelées **paramètres réels** ou **paramètres effectifs** ou **arguments**.

  Par exemple, dans l'appel suivant, deux arguments, une variable (`s1`) et une constante chaîne de caractères, sont passés : `strcpy( s1 , "Le langage C");`

[1] beurk ! toute utilisation de variables globales devra être parfaitement justifiée

- Du coté appelé, les variables spécifiées sont appelées **paramètres formels** ou **paramètres**.

- Lors de l'appel de la fonction, la liste des paramètres réels est mise en correspondance avec la liste des paramètres formels.

- Le mode de transmission entre ces deux contextes est effectué **par valeur** : la fonction travaille donc sur une **copie** des paramètres réels.

Attention :

- En compilation classique, le nombre et le type des paramètres effectifs ne sont pas contrôlés. Cette propriété est parfois utilisée.

- L'ordre d'évaluation des paramètres n'est pas défini dans le langage. Ainsi dans l'instruction suivante :

```
toto ( ++i , tab[i] );
```

  est-ce `i` ou `i` incrémenté de 1 qui est l'index de `tab` ?

7.3 L'appel d'une fonction

- L'appel d'une fonction est réalisé en signifiant le nom de la fonction suivi de la liste des paramètres réels.

- L'appel de la fonction suppose que celle ci ait été définie ou déclarée auparavant. Dans le cas contraire C suppose que la valeur retournée est du type `int` et aucune erreur n'est signalée ...

Exemple :

```
b = theta ( 'a' , 180 , v );
pi2 = 2 * pi();
cls();
```

7.4 Définition classique d'une fonction

```
classe  type  identificateur( liste_parametres_formels )
declaration_parametres_formels
bloc_des_instructions
```

- **classe** : Par défaut, la fonction est visible dans tous les modules de l'application.
  - `extern` : permet de définir une fonction qui est définie dans un autre module de l'application.
  - `static` : la fonction n'est visible que dans le module dans lequel elle est définie.

- **type** : (facultatif) type de la valeur retournée par la fonction. Par défaut c'est le type `int` qui est retourné.

  Le type `void` précise que la fonction ne retourne pas de valeur.

- **identificateur** : identificateur de la fonction.

- **liste_parametres_formels** : c'est la liste des identificateurs des paramètres formels de la fonction. Ils ont une durée de vie limitée à la durée d'exécution de la fonction et une visibilité réduite au bloc de la fonction.

Exemple :

```
float  theta( cas , angle , vitesse )
char  cas;           /* declaration des parametres */
int   angle;
float vitesse;
{                    /* bloc de la fonction */
instruction1;
...
}
```

7.5 Déclaration classique d'une fonction

Avant utilisation une fonction doit, comme les autres objets C, être déclarée.

Cette déclaration permet de préciser uniquement le type de la valeur retournée par la fonction. Le compilateur C ne vérifiant pas le type et le nombre de paramètres formels, il est inutile de les faire figurer.

Il existe des programmes qui vérifient la cohérence des appels des fonctions (`/usr/bin/lint`).

- **Syntaxe :**

```
classe type identificateur( ) ;
```

`classe` et `type` : voir définition classique de fonction.

- **Exemple :**

```
float  theta( );
```

7.6 Définition ANSI d'une fonction

- **Syntaxe :**

```
classe  type  identificateur( liste_typee_parametres_formels )
   bloc_des_instructions
```

`classe` et `type` : voir définition classique de fonction.

`liste_typee_parametres_formels` : c'est la liste de déclaration des paramètres formels de la fonction.

- **Exemple :**

```
float  theta( char cas , int angle , float vitesse )
{                    /* bloc de la fonction */
instruction1;
...
}
```

  - **const** : si l'on utilise `const` en tant que modificateur dans la liste des paramètres d'une fonction pour un paramètre pointeur, `const` interdit à la fonction de modifier le pointeur ou la variable pointée.

    Exemples :

    `int affiche( const char *msg );`

    La fonction `affiche` ne peut pas modifier le contenu de la chaîne pointée par `msg`, mais elle peut modifier la valeur du pointeur `msg`.

    `void echange( int const *v1 , int const *v2 );`

    `v1` et `v2` sont des pointeurs constants vers des entiers. On ne peut donc pas modifier la valeur du pointeur, mais on peut modifier la valeur pointée par ces derniers.

7.7 Déclaration ANSI d'une fonction

- **Syntaxe :**

```
classe type identificateur( liste_typee_parametres_formels ) ;
```

`classe` et `type` : voir définition classique de fonction.

On peut employer le type `void` pour une fonction n'ayant pas de paramètres ou ne retournant aucune valeur.

- **Exemple :**

```
float  theta( char cas , int angle , float vitesse ) ;
double pi(void);
void   cls( void ) ;
```

Une déclaration de fonction peut toujours être faite en style classique.

7.8 Les prototypes

On nomme **prototype**, une déclaration complète de fonction qui comprend donc :

- Le type de la valeur retournée par la fonction
- la liste de déclaration des paramètres formels, qui peuvent avoir un des formats suivants :
  - **type** : seul le type est utilisé par le compilateur pour mettre en place les conversions nécessaires

    Exemple : `float theta( char , int , float );`
  - **type nom_parametre** : les identificateurs `nom_parametre` sont purement fictifs et ne servent qu'à donner au prototype une forme comparable à l'en-tête de la fonction.

    Exemple : `int somme(int a , int b);`
  - ... (ellipse) pour un nombre variable de paramètres dont le type ne sont pas connus par la fonction appelée.

    Exemple : `int  printf( const char *format , ... );`

Le prototype est utilisé par le compilateur pour mettre en place les conversions nécessaires et vérifier que les paramètres réels correspondent bien en nombre et en type aux paramètres formels.

En l'absence de prototype le compilateur mettra en œuvre les conversions par défaut.

Il est possible de passer un `char` ou un `float`, par exemple, en paramètre sans qu'ils soient convertis respectivement en `int` et `double` dans le cas d'une déclaration classique (conversions par défaut).

7.9 Passage par référence

Ce mode de passage, contrairement à d'autres langages, n'existe pas en C. Le seul mode de passage des paramètres est le mode de passage "par valeur". De plus, une fonction ne peut retourner qu'une seule valeur.

Si l'on désire écrire une fonction qui puisse :

- nous fournir plusieurs résultats,
- nous fournir un résultat dans un de ses paramètres réels,
- modifier un paramètre d'entrée,

il faut :

1. soit définir une structure pour les valeurs de retour de la fonction,

   Exemple de déclaration d'une fonction qui doit nous fournir les valeurs min et max d'un tableau d'entiers :

```
int *min_max(int tab[], int dim);
/* retourne un tableau de 2 entiers contenant le min et le max  */
/* entraine un probleme d'allocation des valeurs retournees ... */
```

2. soit fragmenter les fonctions, de sorte qu'elles n'aient qu'une seule valeur de retour chacune,

   ```
   int min(int tab[], int dim);
   int max(int tab[], int dim);
   ```

   ou

   ```
   int min_max(int tab[], int dim, int flag);
   /* si flag = 0     min_max()  retourne la valeur min */
   /* si flag = 1     min_max()  retourne la valeur max */
   ```

3. soit passer par des pointeurs : on transmet un pointeur sur l'objet en question : le paramètre formel fait alors référence à la même position mémoire que l'argument.

   Si la fonction change la valeur contenue à cette adresse, elle change aussi la valeur de l'argument passé.

   ```
   void min_max(int tab[], int dim, int *min, int *max);
   ```

Les deux premières solutions ne sont pas élégantes et c'est la troisième solution qui est utilisée par les programmeurs C.

Exemple : échange du contenu de 2 variables

```
#include <stdio.h>

void echange_rien(int a , int b)
{  int temp;
   temp = a; a = b; b = temp;
}

void echange_bien(int *pa , int *pb)  /* echange de 2 entiers */
{  int temp;
   temp = *pa; *pa = *pb; *pb = temp;
}

main(void) {
  int x = 10 , y = 20;

  echange_rien( x , y );
  printf("x = %d   y = %d\n", x, y);  /* affichage:  x = 10  y = 20 */

  echange_bien( &x , &y );
  printf("x = %d   y = %d\n", x, y);  /* affichage:  x = 20  y = 10 */
}
```

7.10 L'instruction `return`

Cette instruction :

1. permet de renvoyer la valeur précisée en argument à l'instruction appelante.
2. provoque une sortie immédiate du bloc principal de la fonction.

Exemple :

```
return 12;
return ;
return ( a > b ) ? a : b ;
```

7.11 Pointeurs sur des fonctions

Le nom seul d'une fonction est un **pointeur constant** sur la première instruction exécutable de celle ci.

Ainsi : `int ( *ptf )(int , int);`

définit `ptf` comme étant un pointeur sur une fonction (ayant 2 arguments de type `int`) retournant un entier.

Si l'on déclare une fonction comme : `int somme( int n1 , int n2 );`

un appel indirect à la fonction `somme` peut se faire de la manière suivante :

```
ptf = somme;

result = (*ptf)(5, 3);    /* result contiendra la valeur 8 */

result = ptf(5, 3);       /* result contiendra la valeur 8 */

result = (*somme)(5, 3);  /* result contiendra la valeur 8 */
```

Exemple :

```
#include <stdio.h>

int addition(int n1, int n2) {
  return n1 + n2;
}

int soustraction(int n1, int n2) {
  return n1 - n2;
}
```

```
int multiplication(int n1, int n2) {
  return n1 * n2;
}

int division(int n1, int n2) {
  return n1 / n2;
}


main(void) {
  int (*ptf[])(int , int) = {addition, soustraction,
                             multiplication, division };
  /* ptf est un tableau de pointeurs sur des fonctions qui
     retournent un entier. Ce tableau est initialise avec
     les adresses des 4 fonctions definies ci dessus       */
  char operation, tabop[]="+-*/";
  int nbre1, nbre2, i;

  printf("Nombre 1 : ");
  scanf("%d", &nbre1); fflush(stdin);
  printf("Nombre 2 : ");
  scanf("%d", &nbre2); fflush(stdin);
  do {
    printf("operation ( +, -, * ou /) : ");
    scanf("%c", &operation); fflush(stdin);
    for (i=0; i<4; i++)
      if (tabop[i] == operation)
        break;
  } while ( i==4 );

  printf("%d %c %d = %d\n", nbre1, operation, nbre2,
                            (*ptf[i])(nbre1, nbre2) );
  /* deuxieme possibilite d'appel indirect */
  printf("%d %c %d = %d\n", nbre1, operation, nbre2,
                            ptf[i](nbre1, nbre2) );

  return 0;
}
/*-- resultat de l'execution --------------------------------
Nombre 1 : 12
Nombre 2 : 34
operation ( +, -, * ou /) : q
operation ( +, -, * ou /) : +
12 + 34 = 46
12 + 34 = 46
------------------------------------------------------------*/
```

7.12 Arguments de `main()`

Les arguments, de la ligne de commande qui lance l'exécution d'un programme, sont fournis au programme sous la forme d'un tableau de pointeurs. Pour pouvoir les utiliser, il faut déclarer la fonction `main` de la façon suivante :

```
main(int argc , char *argv[])
```

`argc` contient le nombre d'arguments de la ligne de commande.

`argv` est un tableau de pointeurs sur les arguments de la ligne de commande shell.

`argv[0]` pointe sur le nom de la commande, il est donc toujours le nom du programme lui même (et donc `argc` est toujours au moins égal à 1).

`argc` et `argv` sont des noms conventionnels. La norme précise que `argv[argc]` doit être un pointeur `NULL`.

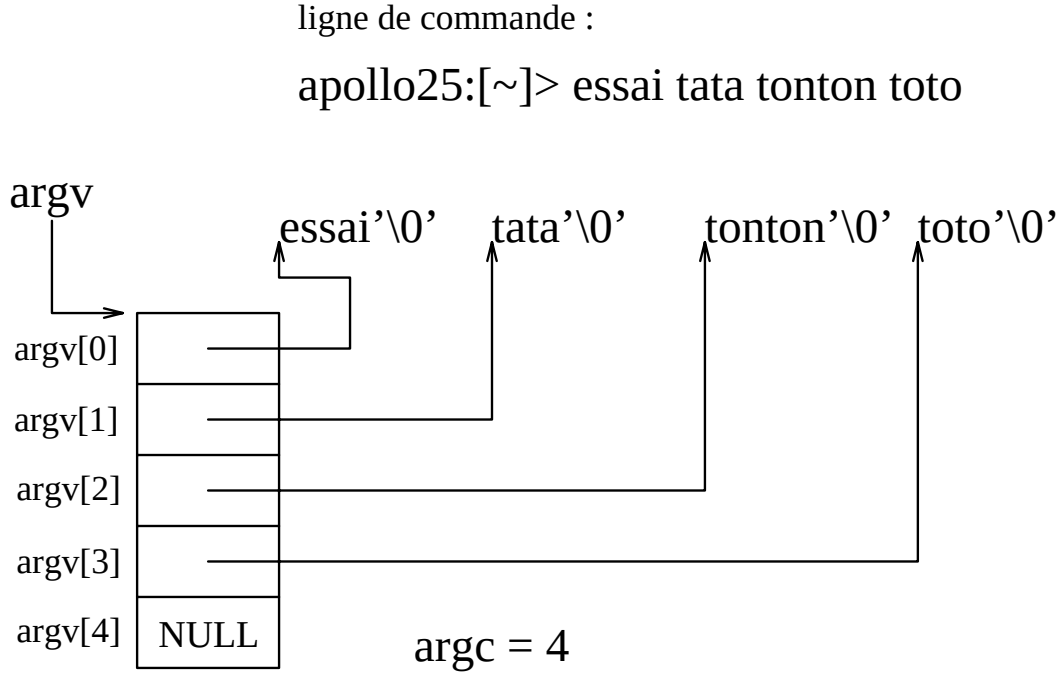


Figure 7.1: Arguments de la fonction `main`

Exemples :

```
/* affiche les arguments de la ligne de commande, version tableau */

int main(int argc , char *argv[])
{ register int i;
  for ( i=0 ; i < argc ; i++ )
     printf("%s\n",argv[i]);
}
```

```
/* affiche les arguments de la ligne de commande, version pointeur */

int main(int argc,char **argv)
{                /*    **argv  equivalent: *argv[]  equivalent: argv[][]  */
  while ( argc--)
     printf("%s\n",*argv++);
}
```

7.13 Fonctions retournant un pointeur

Une erreur courante (chez les débutants) est de retourner l'adresse d'un tableau de classe `auto`. En effet, les variables de cette classe sont stockées dans la pile et n'existent plus après l'appel de la fonction.

Exemple :

```
/* upcasestr()  */
#include <stdio.h>
#include <ctype.h>

/* Retourne la chaine str en majuscule  */
/* ********* NE FONCTIONNE PAS ******** */
char *upcase_str(char *str)
{
  char temp[256];
  char *ptr = temp;

  for ( ; (*ptr = *str) != '\0' ; ptr++ , str++ )
    if ( islower(*str) )
      *ptr = toupper(*str);
  return(temp);
}

main(){
  char ch[] = "C is beautiful";
  char *ch1, *ch2;

  ch1 = upcase_str(ch);
  ch2 = upcase_str("Bizarre");
  printf("%s and %s\n",ch1, ch2);
}
```

L'exécution de ce programme a toutes les chances de ne rien afficher, ou d'afficher n'importe quoi. En effet le tableau local `temp` est, par défaut, de classe `auto` et par conséquence il est détruit à la sortie de la fonction. La fonction appelante reçoit donc en retour l'adresse **d'une variable qui n'existe plus**.

- **Première solution**

La première solution consiste à mettre le tableau local `temp` de classe `static`. Sa durée de vie est donc maintenant celle du programme.
On remplace donc la ligne : `char temp[256];` par : `static char temp[256];`

Le programme modifié affiche maintenant :

`BIZARRE and BIZARRE` au lieu de : `C IS BEAUTIFUL and BIZARRE`

Dans le programme ci dessus, `ch1` et `ch2` désigne la même adresse en mémoire, et donc le deuxième appel à la fonction `upcasestr` écrase le résultat du premier appel.

Avantage :

- cette méthode ne réserve qu'une fois le tableau en mémoire.

Inconvénient :

- il faut recopier le résultat dans une autre variable.

le programme principal, pour bien fonctionner, doit s'écrire maintenant :

```
main(){
  char ch[] = "C is beautiful";
  char ch1[80], ch2[80];

  strcpy(ch1 , upcase_str(ch) );
  strcpy(ch2 , upcase_str("Bizarre") );
  printf("%s %s\n",ch1, ch2);
}
```

- **Deuxième solution**

Cette solution consiste à faire une allocation dynamique du tableau `temp`.

Le fonction `upcase_str` s'écrit maintenant :

```
char *upcase_str(char *str )
{
  char *temp, *ptr;

  temp = (char *) malloc( strlen( str ) + 1);
  for ( ptr = temp ; (*ptr = *str) != '\0' ; ptr++ , str++ )
    if ( islower(*str) )
      *ptr = toupper(*str);
  return(temp);
}
```

Avantage :

- le résultat retourné est correct, car la fonction retourne bien une adresse valide.

Inconvénients :

- chaque appel à la fonction `upcase_str`, alloue une nouvelle zone mémoire : risque de saturation de la mémoire.
- il faut libérer la mémoire allouée (c.f. `free` page 147). Ceci introduit une dissymétrie entre l'allocation, qui se fait dans une fonction et la libération qui se fait en dehors de celle-ci. Par ailleurs, la libération de cette mémoire n'est pas toujours possible, dans le cas ou l'adresse de retour est perdue comme dans l'exemple ci dessous :

```
for ( i=0 ; i<1000 ; i++)
   puts( upcase_str("Bonjour ") );
```

- **Troisième solution**

Cette solution consiste à travailler avec les paramètres de la fonction.

La solution (triviale) qui consisterait de travailler avec un seul paramètre qui fournirait en entrée la chaîne à convertir, et en sortie la chaîne traitée n'est pas acceptable, car cette solution détruirait la chaîne d'origine.

Il faut donc que la fonction renvoie par un autre paramètre la chaîne traitée.

```
/* upcasestr() */
#include <stdio.h>
#include <ctype.h>

/*  Retourne dans dest la chaine str en majuscule  */
void upcase_str(char *dest, char *str )
{
  for ( ; (*dest = *str) != '\0' ; dest++ , str++ )
    if ( islower(*str) )
      *dest = toupper(*str);
}

main(){
  char ch[] = "C is beautiful";
  char ch1[80], ch2[80];

  upcase_str(ch1, ch);
  upcase_str(ch2, "Bizarre");
  printf("%s and %s\n",ch1, ch2);
}
```

- avantage :

C'est à l'appelant de fournir et d'allouer la zone mémoire recevant le résultat de la conversion.

- inconvénient :

Du fait que la fonction ne renvoie pas de valeur, cette solution est parfois plus difficile et plus lourde à mettre en œuvre.

- **Quatrième solution**

Celle ci reprend donc la troisième solution et retourne aussi l'adresse de la chaîne traitée.

```
/* upcasestr() */
#include <stdio.h>
#include <ctype.h>

/*  Retourne la chaine str en majuscule  */
char *upcase_str(char *dest, char *str )
{
  char *ptr = dest;

  for ( ; (*dest = *str) != '\0' ; dest++ , str++ )
    if ( islower(*str) )
      *dest = toupper(*str);
  return(ptr);
}

main(){
  char ch[] = "C is beautiful";
  char ch1[80], ch2[80];

  /* utilisation comme une fonction */
  printf("%s", upcase_str(ch1, ch));

  /* utilisation comme une "procedure" */
  (void) upcase_str(ch2, "Bizarre");
  printf(" and %s\n", ch2);
}
/*-- resultat de l'execution ----------------------------
C IS BEAUTIFUL and BIZARRE
-------------------------------------------------------------*/
```

C'est cette solution qui est souvent utilisée, comme par exemple, pour les fonctions de la librairie standard (`strcpy, gets` ...).

- avantage :

On peut utiliser cette fonction comme une "procédure" ou comme une fonction.

7.14 Les listes variables d'arguments

Il est possible, en langage C, d'écrire des fonctions, comme `printf`, dont le nombre d'arguments n'est pas fixé lors de la définition.

7.14.1 L'ellipse ...

Les points de suspension (au nombre de trois, sans espace intermédiaire) permettent d'indiquer un nombre variable d'arguments, ou encore des arguments ayant des types variables.

Exemple de déclaration :

```
vois erreur(int n, char *fmt, ...);
```

Cette déclaration dit que la fonction `erreur` est défini de telle manière que les appels doivent fournir au moins deux arguments, un de type `int` et un de type `char` mais qu'ils peuvent fournir des arguments supplémentaires.

Exemples d'appels :

```
erreur( 3, "Erreur: mission impossible");
erreur( valeur, "%s %d\n", message, errno);
```

Remarque : la fonction devra obligatoirement avoir au moins un argument connu avant le ... et donc il n'est pas possible d'écrire la déclaration suivante :

```
void erreur(...);  /* declaration fausse */
```

7.14.2 Accès aux paramètres

Les macros `va_start`, `va_arg` et `va_end` [2] permettent d'accéder aux paramètres des fonctions acceptant un nombre variable d'arguments.

- **Syntaxe :**

```
#include <stdarg.h>
void va_start(va_list ap, param);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

- **Description :**
  - la macro `va_start` fait pointer `ap` sur le premier argument variable fourni à la fonction. `param` étant le nom du dernier paramètre nommé.
  - la macro `va_arg` renvoie le premier argument variable et fait pointer `ap` sur l'argument suivant. `type` est le type de l'argument qui va être lu et `va_arg` génère une expression de ce même type.

[2] `va` comme **v**ariable **a**rgument

- la macro `va_end` remet tout en état normal avant le retour à la fonction appelante.

- **Remarque :** le problème à résoudre consiste à définir une méthode de calcul du nombre d'arguments réels de la fonction appelante, pour que la fonction appelée puisse déterminer le nombre d'arguments à traiter.

  Par exemple :

  - Le premier argument peut être le nombre de paramètres réels.
  - un argument devra avoir une valeur particulière (0 par exemple) afin d'indiquer la fin des arguments variables (c.f. exemple 1).
  - Le premier argument doit être structuré d'une telle façon que la fonction appelée doit pouvoir compter les arguments variables (c.f. exemple 2).

- **Exemple 1 :**

```
/* calcule la somme de n entiers     */
/* le dernier argument doit etre a 0 */
#include <stdio.h>
#include <stdarg.h>


int somme(int n1, ...) {
  va_list  pa;
  int som, n;

  som = n1;
  va_start(pa, n1);
  while( (n = va_arg(pa, int)) != 0)
     som = som + n;
  va_end(pa);
  return som;
}

main(void) {
  printf("1 + 3 + 5 + 7 + 9 = %d\n", somme(1,3,5,7,9,0));
  printf("1 + 1 = %d\n", somme(1,1,0));
  return 0;
}
/*-- resultat de l'execution ---------------------------
1 + 3 + 5 + 7 + 9 = 25
1 + 1 = 2
------------------------------------------------------*/
```

- **Exemple 2 :**

```
#include <stdio.h>
#include <stdarg.h>
```

```
void myprintf(char *format, ...) {
  va_list  pa;
  int n;
  char *s, c;
  float f;

  va_start(pa, format);
  while (*format != '\0') {
    if ( *format == '%' ) {
      switch (*++format) {
        case '%' : putchar('%'); break;
        case 'c' : /* affichage d'un caractere */
                   c = va_arg(pa, char);
                   putchar(c);
                   break;
        case 'd' : /* affichage d'un entier */
                   n = va_arg(pa, int);
                   printf("%d", n);
                   break;
        case 'f' : /* affichage d'un float */
                   f = va_arg(pa, double);    /* !!!!! */
                   printf("%f", f);
                   break;
        case 's' : /* affichage d'une chaine */
                   s = va_arg(pa, char *);
                   for ( ; *s != '\0'; s++ )
                     putchar( *s );
                   break;
      } /* end switch */
    }
    else
      putchar( *format );
    format++;
  }
  va_end(pa);
}

main(void) {

  myprintf("float = %f\n", (float) 1.2345);
  myprintf("int = %d   char = %c   Chaine = %s\n",
             123, 'A', "C is beautiful !" );
  return 0;
}
/*-- resultat de l'execution ---------------------------------
float = 1.234500
int = 123   char = A   Chaine = C is beautiful !
------------------------------------------------------------*/
```

7.15 La récurrence

C'est une technique de programmation qui implique plus d'une activation de la fonction à un moment donné. Il y a deux types de récurrence, la récurrence directe et la récurrence indirecte.

1. **La récurrence directe** a lieu quand une fonction s'appelle elle même. Chacune des activations de cette fonction a sa propre zone mémoire réservée (pile) pour les variables de classe *auto* et les paramètres.

   A un certain moment, cependant, la fonction doit cesser de s'appeler elle-même sous peine de remplir complètement la pile. Un appel récursif doit donc être conditionnel : quand une certaine condition est satisfaite, la fonction doit cesser de s'appeler elle-même et doit terminer son exécution. Quand la fonction se termine, le contrôle revient à l'activation précédente. Ce processus continue jusqu'à l'achèvement de toutes les activations.

   Exemple très classique :

```
/* factorielle */
#include <stdio.h>

long facto( long n )
{
  if ( n == 1 )
     return 1;
  else
     return n * facto( n - 1 );
}

main(){

  printf(" 8! = %ld\n", facto(8) );
}
```

2. **La récurrence indirecte** a lieu quand une fonction est appelée via une série d'appels de fonctions qu'elle a elle même entamée.

   Par exemple, `f1()` appelle `f2()` et `f2()` appelle `f1()`

Chapitre 8

Les types dérivés

8.1 Les structures (`struct`)

Les tableaux permettent de désigner sous un même nom un ensemble de valeurs de même type. Les structures permettent de regrouper des objets de types hétérogènes.

Une variable de type structure est une variable composée de champs, eux-mêmes déclarés d'un certain type.

8.1.1 Déclaration de type structure

Une déclaration de type structure ne définit aucune variable, elle permet de définir un modèle de structure.

Syntaxe :

```
struct identificateur_de_type
       { type identificateur_de_champ;
         type identificateur_de_champ;
         ...
       };
```

Exemple :

```
struct s_fiche {
       char nom[30];
       char sexe;
       int  numero;
};
```

On peut donc définir des variables ayant ce type :

```
/* definition de etat_civil de type s_fiche */
struct s_fiche etat_civil;

/* definition et initialisation */
struct s_fiche fiche = {"Ritchie",'M',"65"};
```

```
/* tableau de 100 structures de type s_fiche */
struct s_fiche fichier[100];
```

La déclaration d'une variable structure peut suivre immédiatement la définition du type :

```
struct identificateur_de_type
       { type identificateur_de_champ;
         type identificateur_de_champ;
         ...
       } identificateur_de_variable ... ;
```

Exemple :

```
struct t_fiche {
       char nom[30];
       char sexe;
       int  numero;
} etat_civil , fiche;
```

Les champs d'une structure peuvent être de n'importe quel type (sauf elle même ou une fonction).

Attention : pour connaître la taille d'une variable de type structure, il faut impérativement utiliser l'opérateur `sizeof` (cf. 3.6 page 29).

8.1.2 Utilisation des champs d'une structure

Pour faire référence à un champ particulier de la structure, on accole le nom de la structure concernée avec le nom du champ visé, et on insère le caractère . (point) entre les 2 termes.

Exemple :

```
  strcpy(etat_civil.nom , "Thompson") ;
  etat_civil.sexe = 'M' ;
```

On peut aussi faire référence au champ à l'aide d'un pointeur :

```
/* definition d'une variable pointeur sur une structure de type s_fiche */
struct s_fiche *membre;
  ...
/* acces au champ nom */
   membre->nom      /* equivalent :  (*membre).nom   */
```

l'opérateur de pointeur de structure -> (symbole - (moins) suivi du symbole > (supérieur)) pointe vers un membre particulier de la structure.

Il est possible d'affecter à une structure le contenu d'une autre structure définie avec le même modèle.

```
   struct s_fiche fiche1 , fiche2;

   fiche1 = fiche2;
```

Exemple :

```
#include <stdio.h>

struct s_date {
   int jour;
   int mois;
   int an;
};

struct s_fiche {
   char nom[20];
   int  numero;
   struct s_date naissance;
 };

void afficher( struct s_fiche f );
struct s_date SaisirDate( void );
void SaisirFiche( struct s_fiche *f );

/*----------- programme principal -------------*/
main(void) {
  struct s_fiche fiche;

  SaisirFiche(&fiche);
  afficher(fiche);
  return 0;
}

/*-------------------------------------*/
void afficher( struct s_fiche f ) {

   printf("Nom : %s\n", f.nom );
   printf("Numero : %d\n", f.numero);
   printf("Date de naissance : %d/%d/%d\n",
               f.naissance.jour, f.naissance.mois,
               f.naissance.an );
}
```

```
/*-----------------------------------*/
struct s_date SaisirDate( void ) {
    struct s_date date;

    printf("Jour ? ");
    scanf("%d", &date.jour); fflush(stdin);
    printf("Mois ? ");
    scanf("%d", &date.mois); fflush(stdin);
    printf("Annee ? ");
    scanf("%d", &date.an); fflush(stdin);
    return date;
}

/*-----------------------------------*/
void SaisirFiche( struct s_fiche *f ) {

    printf("Nom ? ");
    gets( f->nom );
    printf("Numero ? ");
    scanf("%d", &f->numero); fflush(stdin);
    f->naissance = SaisirDate();
}
```

8.1.3 Structures récursives (ou autoréférentielles)

Ce sont des structures se référant à elles-mêmes. Elles permettent de définir des listes, des arbres ...

Il sera souvent plus naturel d'utiliser avec ce genre de structure, des fonctions récursives pour l'insertion, la recherche ou l'affichage d'un élément.

Exemple de définition d'une arborescence binaire :

```
struct s_fiche {
       char *nom;
       int  numero;
       struct s_fiche *pere , *mere;
};
```

`pere` et `mere` sont des pointeurs sur une structure de type `s_fiche`.

Exemple :

```
/* manipulation d'une liste chainee de caracteres */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct s_chaine {
```

```
    char c;
    struct s_chaine *suiv;
} CHAINE;

#define MALLOC( t ) (t *) malloc( sizeof( t ) )
#define NIL  (CHAINE *) 0

CHAINE *make_str( CHAINE *str, char c );
int length_str( CHAINE *str );
void write_str( CHAINE *str );

main(void) {
  CHAINE *tete = NIL;
  char *pt = "Programmer: c'est ecrire une instruction juste et... programmer";

  while ( *pt )
    tete = make_str(tete, *pt++);

  printf("Longueur : %d\n", length_str(tete));
  write_str( tete );
  putchar( '\n' );

  /* liberation de la liste */
  { CHAINE *ptr=tete->suiv;

    while ( tete != NIL) {
      free( tete );
      tete = ptr;
      ptr = tete->suiv;
    }
  }
  return 0;
}

/*--------------------------------------*/
CHAINE *make_str(CHAINE *str, char c) {
  if ( str == NIL) {
    str = MALLOC(CHAINE);
    str->c = c;
    str->suiv = NIL;
  }
  else
    str->suiv = make_str(str->suiv, c);
  return str;
}

/*--------------------------------------*/
int length_str(CHAINE *str) {
  return (str == NIL) ? 0 : 1 + length_str(str->suiv);
}
```

```
/*-------------------------------------*/
void write_str(CHAINE *str) {
  if ( str != NIL ) {
    putchar(str->c);
    write_str(str->suiv);
  }
}
/*-- resultat de l'execution ------------------------
Longueur : 63
Programmer: c'est ecrire une instruction juste et... programmer
-----------------------------------------------------*/
```

8.1.4 const et les structures

Le modificateur `const` peut être aussi appliqué à une structure ou à l'un des membres de la structure.

Exemple :

```
struct date { int j, m, a; };
const struct date hier = {1, 1, 1993};
struct { const int n1; int n2; } essai = {1, 2};

hier.j = 2;     /* erreur : hier.j n'est pas une lvalue */
essai.n1 = 10;  /* erreur : essai.n1 n'est pas une lvalue */
essai.n2 = 20;  /* OK */
```

8.2 Structure de bits

Dans une structure, il est possible de définir un champ comme une suite continue de bits.

Elles sont utiles pour les applications orientées "machine", elles permettent d'isoler et de traiter des bits à l'intérieur d'un mot, plutôt que de manipuler des "masques" avec des opérateurs de traitement de bits.

Syntaxe :

```
struct identif_type {
       type  [id_champ] [ : nbre_bits] ;
       type  [id_champ] [ : nbre_bits] ;
       ...
} [identif] ;
```

- `type` : `int` ou `unsigned`. Les champs se comportent comme des variables du type spécifié et peuvent intervenir dans les expressions arithmétiques comme des entiers du type `int` ou `unsigned int`.

- `id_champ` : identificateur du nom du champ. Facultatif, il signifie dans ce cas, que l'on doit sauter le nombre de bits correspondants à `nbre_bits`.
- `nbre_bits` : Nombre de bits constituant le champ de nom `id_champ`. Si `nbre_bits` est absent, la longueur du champ est la longueur d'un `int`

Exemple :

```
 struct machine {
        unsigned  nb_bits : 2;
        unsigned  nb_stop : 1;
        unsigned  parite  : 2;
        unsigned  vitesse : 3;
 } *rs_232;

rs_232 = 0x3f8;
rs_232->vitesse = 5;
```

Attention :

- Si une valeur dépasse les possibilités d'un champ, sa valeur est tronquée.
- La norme ne précise pas si la description d'un champ de bits se fait en allant des poids faibles vers les poids forts ou inversement.
- Si une structure de bits occupe plusieurs octets, l'ordre dans lequel ces derniers sont décrits dépend de l'implémentation.

8.3 L'union (`union`)

Une union est comparable à une structure sauf qu'elle permet de définir plusieurs variables stockées au même endroit en mémoire. Cela peut s'avérer utile pour interpréter de plusieurs façons une zone mémoire.

Le principe de déclaration de type union ou de définition est le même que pour les structures. Le mot réservé `struct` est remplacé par `union`.

Exemple :

```
/* definition d'un type union */
union  essai {
       int  a;
       char tab[4];  /* sur une machine 32 bits */
       };

/* definition d'une variable zone de type union essai */
union essai zone;
```

C va allouer suffisamment d'espace en mémoire pour le stockage du plus grand des membres de l'union.

Au contraire d'une structure, les variables `zone.a` et `zone.tab` occupent le même emplacement en mémoire. Par conséquent, écrire dans l'une écrira dans l'autre.

On accède aux membres d'une union tout comme on accède aux membres d'une structure.

Exemple :

```
/* illustration des structures de bits et des unions */

#include <stdio.h>

main(void) {
  struct s_float {
    unsigned signe : 1;
    unsigned exposant : 8;
    unsigned mantisse : 23;
  };

  union {
    float f;
    struct s_float sf;
  } ieee;

  float fl, mant;
  int signe, exp;

  ieee.f = 12.345678;

  printf("signe    : %d\n", ieee.sf.signe);
  printf("exposant : %d\n", ieee.sf.exposant);
  printf("mantisse : %d\n", ieee.sf.mantisse);

  exp = 1 << (ieee.sf.exposant - 127);
  mant = ( float) ieee.sf.mantisse  / ( 1 << 23);
  signe = (ieee.sf.signe == 1) ? -1 : 1;

  fl = signe * exp * ( 1 + mant );
  printf("%d * 2^%d * (1 + %f) = %f\n", signe, exp, mant, fl);

  return 0;
}
/*-- resultat de l'execution ---------------------------------
signe    : 0
exposant : 130
mantisse : 4556774
1 * 2^8 * (1 + 0.543210) = 12.345678
-------------------------------------------------------------*/
```

8.4 Enumération (enum)

Une énumération est une liste de valeurs entières constantes. Le compilateur affecte la valeur 0 au premier élément de la liste, 1 au deuxième et ainsi de suite.

On peut affecter une valeur particulière à un élément de la liste; dans ce cas la valeur suivante aura la valeur `+ 1`.

Une énumération permet de définir des constantes entières plus facilement que par une succession de `#define` (cf. 9.2 page 94).

Syntaxe :

```
enum [ id_type ] { identif [ = constante_entiere ] , ... } [ident_var] ...;
```

- `id_type` : (facultatif) permet de donner à l'ensemble un nom de type.
- `identif` : c'est le nom d'une constante à laquelle on peut affecter la valeur de `constante_entiere`.

Exemple :

```
/* definition d'un type enumere couleur */
enum couleur { rouge , bleu , vert , jaune};

/* definition de variables de type enumeration */
enum couleur arc_en_ciel;

enum { juillet = 7 , aout , septembre } vacances;

enum operation {
       addition = '+',
       soustraction = '-',
       multiplication = '*',
       division = '/'
} operateur;

/* utilisation */
arc_en_ciel = vert;
if (arc_en_ciel == bleu ) break;

operateur = (enum operation) getchar();
switch (operateur) {
  case addition :
  ...
}
```

8.5 Définition de nouveaux types (typedef)

La déclaration `typedef` permet de définir de nouveaux noms de types, reconnus par la suite.

Cette définition consiste à faire suivre le mot réservé `typedef` par une construction ayant la même syntaxe qu'une définition de variable. Ces nouveaux types sont ensuite utilisables comme un type de base.

Exemple :

- `typedef int entier;`

  `entier` est un synonyme de `int`.

- `typedef int tab[10];`

  Les déclarations suivantes sont équivalentes :

  ```
  int x[10];          tab x;
  ```

- `typedef int (*p_f)();`

  Déclare un type nommé `p_f` qui est un pointeur sur une fonction qui retourne un entier.

Le constructeur `typedef` est très souvent utilisé avec les définitions de structures :

```
struct fiche {
         char nom[30];
         int  age;
};

typedef struct fiche fiche;

fiche adherent;   /* equivalent :   struct fiche adherent;  */
```

Chapitre 9

Le préprocesseur

C'est la première phase d'un processus de compilation. Elle est lancée automatiquement par la commande `cc`. Il est chargé d'effectuer certaines actions préliminaires avant la compilation.

Son activité est uniquement une transformation de texte afin de produire un nouveau fichier.

Les tâches du préprocesseur sont :

- élimination des commentaires,
- l'inclusion de fichier source,
- la substitution de texte,
- la définition de macros,
- la compilation conditionnelle.

Une ligne commençant par le caractère # (dièse) permet de définir une directive au préprocesseur.

Le préprocesseur peut être lancé indépendamment en exécutant le programme `/lib/cpp` sur un fichier texte pouvant être autre chose qu'un programme C.

9.1 La directive `#include`

- **Syntaxe :**

```
#include < nom_fichier >
#include " nom_fichier "
```

La première forme de syntaxe permet d'incorporer un fichier de nom `nom_fichier` se trouvant dans le répertoire `/usr/include`.

La deuxième forme recherche le fichier désigné dans le répertoire courant.

- **Description :**

  Cette directive du préprocesseur permet d'incorporer, avant compilation, le texte figurant dans un fichier. Le texte ainsi incorporé est traité comme s'il se trouvait dans le fichier courant.

  Ce fichier contient en général des déclarations, et a pour extension `.h` (header) (cf. 10.4 page 101).

- **Remarque :** si le fichier se trouve dans un autre répertoire, il faut préciser son chemin absolu.

9.2 La directive `#define`

Elle permet :

- des substitutions de texte,
- la définition de macros C.

9.2.1 Substitution de texte

- **Syntaxe :**

```
#define identificateur texte
```

- **Description :**

  Cette directive permet de définir des pseudo-constantes. Elle remplace chaque occurrence de `identificateur` dans le programme C par la chaîne `texte`.

  L'usage est de mettre les identificateurs des pseudo-constantes en majuscules.

- **Exemple :**

```
#define TAILLE 100
#define SAUT_DE_LIGNE printf("\n")
#define THEN
```

- **Attention :**

  - Ne pas mettre de ; (point virgule) à la fin de la directive, des surprises difficiles en découlent :

```
#define TAILLE 100;  /*  ; de trop  */

...

i = TAILLE - 2;
```

est compilé comme :

```
#define TAILLE 100;  /*  ; de trop  */

...

i = 100;
 - 2;        /* expression valide en C !!! */
```

– Mettre des parenthèses si l'expression est composée :
La pseudo-constante `MAXI` risque de poser d'énormes problèmes :

```
#define TAILLE 10
#define MAXI 2 * TAILLE + 3

int *ptr;
ptr = (int *) malloc( MAXI * sizeof(int) );
```

Ecrite sous cette forme, la fonction `malloc` alloue une taille de 32 octets (`2 * 10 + 3 * sizeof(int)`) au lieu de 92 octets (`23 * sizeof(int)`).
La bonne définition de `MAXI` est `#define (MAXI 2 * TAILLE + 3)`

9.2.2 Définition de macros (ou pseudo-fonctions)

- **Syntaxe :**

```
#define nom_macro(identif_p1 , ... ) texte
```

- **Description :**

Le préprocesseur remplace toute occurrence de `nom_macro` par `texte`. Les paramètres formels `identif_p1` ... sont remplacés par les paramètres effectifs.

- **Exemple :**

```
#define CARRE(X) X*X
#define SOMME(X,Y) X+Y
```

Si dans le programme on trouve :

```
z = CARRE(a);
y = CARRE(c+1);
```

le préprocesseur remplace ces instructions par :

```
z = a*a;
y = c+1*c+1;    /* etonnant !!!  equivalent :  c+(1*c)+1 */
```

Pour éviter cela il faut utiliser des parenthèses :

```
#define CARRE(X) (X)*(X)

y = CARRE(c+1);    /*  --> y = (c+1)*(c+1);  */
z = (short) CARRE(c+1);   /*  --> z = (short) (c+1)*(c+1); */
                          /* equivalent  ((short) (c+1)) * (c+1) */
#define CARRE(X) ((X)*(X))
z = (short) CARRE(c+1);   /*  --> z = (short) ((c+1)*(c+1));  ouf !!! */
```

Un certain nombre de fonctions de la bibliothèque standard sont définies comme des macro-instructions :

```
#define _tolower(c) ( (c)-'A'+'a')
#define tolower(c)  ( (isupper(c)) ? _tolower(c) : (c) )
#define getchar()    getc(stdin)
```

- **Attention :**
  - `nom_macro` est **immédiatement** suivi d'une parenthèse ouvrante.
  - Il ne faut pas utiliser dans les paramètres réels des opérateurs pouvant générer des "effets de bord".

    ```
    #define abs(X) ( ((X) < 0) ? -(X) : (X) )
    ```

    L'appel : `abs(i++);` incrémenterait 2 fois `i` car dans la macro `abs`, `i` est évalué 2 fois.
  - Une macro ne peut pas être utilisée comme paramètre d'une fonction, car il n'y a pas d'adresse de point d'entrée.
  - Une macro peut tenir sur plusieurs lignes si chaque ligne de source est terminée par le symbole `\` (dans la signification UNIX, `\` annule le caractère suivant, ici c'est la fin de ligne qui est annulée).

    Exemple :

    ```
    #define erreur(msg) {                \
         fprintf(stderr, "%s\n", msg);   \
         exit(1);                        \
       }
    ```
  - Ne pas mettre de ; (point virgule) à la fin de la directive.

9.2.3 Suppression d'une définition

- **Syntaxe :**

  ```
  #undef identificateur
  ```

- **Description :**

  Cette directive annule le symbole spécifié par `identificateur`, défini auparavant à l'aide de la directive `#define`

- **Exemple :**

```
#define max(a,b) ( ((a) > (b)) ? (a) : (b) )

z = max(x,y);     /* appel de la macro max */
#undef max
z = max(x,y);     /* appel de la fonction max */
```

9.3 La compilation conditionnelle

Ces directives permettent d'ignorer à la compilation certaines parties du texte source :

```
#if expression_constante
    texte_source
    ...
#else  ou  #elif  expression_constante
    texte_source
    ...
#endif
```

La compilation des lignes suivant la directive `#if` n'a lieu que lorsque le résultat de `expression_constante` est une valeur non nulle.

Sinon, les lignes suivantes sont sautées jusqu'à la prochaine directive `#else` ou `#endif`.

Si l'évaluation de `expression_constante` donne un résultat faux, et s'il existe une directive `#else` qui correspond, les lignes comprises entre les deux directives `#else` et `#endif` sont compilées.

`#elif` est identique à `#else`, sauf que `texte_source` suivant le `#elif` n'est compilé que si `expression_constante` du `#elif` est non nulle.

Les directives `#if` peuvent être imbriquées.

```
#if (((int) ((char)0x80)) <0 )
  #define CHAR_MAX 0x7F
  #define CHAR_MIN 0x80
#else
  #define CHAR_MAX 0xFF
  #define CHAR_MIN 0
#endif
```

On peut également tester si un identificateur a fait ou non l'objet d'une définition :

`#ifdef identificateur` donne la valeur 1 si le symbole spécifié par l'identificateur a été défini auparavant à l'aide de la directive `#define`.

`#ifndef identificateur` donne la valeur 1 si le symbole spécifié par l'identificateur n'a pas été défini auparavant à l'aide de la directive `#define`.

Exemple :

```
#ifdef DEBUG
   printf("Espace restant %d\n",taille);
#endif
```

9.3.1 Les noms prédéfinis

- `__FILE__` : constante chaîne de caractères contenant le nom du fichier source.
- `__LINE__` : constante décimale contenant le numéro de ligne courante du source.
- `__DATE__` : constante chaîne de caractères contenant la date de compilation sous la forme : `mm jj aaaa`
- `__TIME__` : constante chaîne de caractères contenant l'heure de compilation sous la forme : `hh~:mm~:ss`
- `__STDC__` : constante valant 1 pour les implémentations conformes à la norme ANSI.

Chapitre 10

Compilation d'un programme C

NOTA : nous ne verrons ici que l'implémentation UNIX. De façon générale le principe reste identique pour d'autres systèmes, à la syntaxe près (se référer à la documentation du compilateur utilisé).

10.1 Module

Un programme C peut-être découpé en plusieurs fichiers sources, appelés **modules**.

L'usage est de regrouper en un même module les fonctions et les variables ayant trait à un même aspect du projet.

Un objet C (variable ou fonction) n'est défini que dans un seul module et peut-être utilisé dans un autre module.

La conception modulaire va de pair avec la compilation séparée des modules. Ceci évite d'avoir à compiler à nouveau des modules qui n'ont pas subit de modification.

10.2 Compilation

La compilation [1] d'un programme C s'effectue en un ou plusieurs phases, dépendant des options de la ligne de commande utilisées :

- **Le préprocesseur C** : (`/lib/cpp`) cette phase examine toutes les lignes commençant par le caractère # et effectue des manipulations sur le texte source du programme (substitutions de texte, inclusion de fichiers, compilation conditionnelle).
- **Le compilateur** : (`/lib/ccom`) cette phase fabrique le code objet à partir du programme généré par le préprocesseur.
- **Le linker** : (`/bin/ld`) qui assure l'édition des liens entre les différents modules objets pour obtenir un programme exécutable.

[1] on emploie souvent (à tort) le terme de "compilation" pour désigner l'ensemble des différentes étapes intervenant dans le processus d'élaboration d'un exécutable

Ces 3 phases sont lancées automatiquement par la commande `/bin/cc`.

Exemples :

`cc prog.c`
génère un fichier exécutable de nom `a.out` (par défaut)

`cc -o prog prog.c`
génère un fichier exécutable de nom `prog` (option -o)

`cc -o prog prog1.c prog2.c`
génère un fichier exécutable de nom `prog` à partir des 2 fichiers sources.

`cc -c prog3.c`
génère un module objet de nom `prog3.O`.

`cc -o prog prog1.c prog2.c prog3.o`
génère un fichier exécutable de nom `prog` à partir des 2 fichiers sources et du module objet.

`cc -o prog prog.c -lm`
Utilise la librairie `libm.a`

10.3 Autres outils

- `make` : (c.f. page 169) permet à partir d'un fichier descriptif (appelé "makefile") de générer un programme composé de plusieurs modules
- `lint` : permet de vérifier le source d'un programme
- `ar` : gestion des archives
- `cb` : code beautify
- `xdb` : symbolic debugger (c.f. B page 175)
- `prof` : profileur
- `sccs` : (Source Code Control System) gestion et contrôle des versions d'un logiciel.

10.4 Librairies

En C les opérations spécialisées (interface avec le système, les entrées-sorties, le traitement des chaînes de caractères ...) sont effectuées par des fonctions d'une librairie.

Comme tout objet C, une fonction d'une librairie doit être déclarée avant son utilisation.

Ces déclarations sont réunies dans des fichiers d'en-tête (d'extension `.h`) dont on demandera l'inclusion en début de module par le préprocesseur.

Ces fichiers ne comportent que des déclarations, les fonctions elles-mêmes sont contenues dans des fichiers de type librairie de modules objets (d'extension `.a`) où le linker ira les chercher.

Exemples de fichiers headers :

```
#include <stdio.h>   /* Entrees-sorties standard */
#include <math.h>    /* bibliotheque mathematique */
#include <string.h>  /* Traitement des chaines de caracteres */
```

Exemples de fichiers librairie :

```
/usr/lib/libc.a   /* librairie standard C */
/usr/lib/libm.a   /* librairie math */
```

10.5 Librairies partagées

Il est possible grâce à cette technique, d'effectuer l'incorporation d'une fonction externe qu'au moment de l'exécution.

Avantages :

- Economie d'espace disque, le fichier compilé ne contient pas le code exécutable des fonctions externes;
- Economie d'espace mémoire centrale, grâce au partage du code exécutable de cette librairie.

Chapitre 11

La librairie C

Ce chapitre décrit les principales fonctions de la librairie standard ANSI du langage C.

Cette librairie est divisée en plusieurs parties. Chaque sous-section a son fichier d'en-tête qui définit les objets trouvés dans cette section.

Les principaux fichiers d'en-tête standard sont :

- `assert.h` : macro de diagnostic
- `ctype.h` : tests de caractères et conversions majuscules-minuscules
- `float.h` : informations sur les réels
- `limits.h` : valeurs limites des entiers
- `math.h` : fonctions mathématiques
- `setjmp.h` : branchements non locaux
- `signal.h` : traitement des signaux
- `stdarg.h` : fonctions sur les arguments variables
- `stdio.h` : entrées-sorties
- `stdlib.h` : utilitaires
- `string.h` : manipulations de chaîne de caractères
- `time.h` : manipulations de dates et d'heures

11.1 La gestion des fichiers

11.1.1 Généralités

Il existe en C deux niveaux de gestion de fichiers :

- **Gestion de bas niveau (ou de niveau 1) :**

  Il s'agit des fonctions qui permettent un appel direct aux primitives du noyau du système.

  Il n'y a pas de bufférisation : on a accès directement au contenu du fichier à travers les caches du noyau.

  Avantage :

  Rapidité : Les échanges entre le fichier et le processus utilisateur sont directs.

  Fonctions de base :

  Les différentes fonctions de base s'effectuent grâce à un descripteur retourné à l'ouverture ou à la création du fichier.

Nom de la fonction	description
`open`	ouverture
`creat`	création
`close`	fermeture
`read`	lecture
`write`	écriture
`lseek`	positionnement
`chmod`	modification des droits d'accès
`umask`	définition des droits d'accès
`link`	établissement d'un lien
`unlink`	destruction
`chdir`	changement de répertoire
`dup`	duplication de descripteur de fichier

Figure 11.1: Fonctions de base de la gestion de bas-niveau des fichiers

Les fichiers ouverts sont fermés quand le processus se termine.

- **Gestion standard (gestion de haut-niveau ou de niveau 2)**

  Ces fonctions permettent l'accès aux fichiers à travers des flux (`stream`). Un flux est un fichier que l'on manipule à l'aide d'un pointeur sur un objet de type `FILE` qui contient les différents renseignements sur le flux (descripteur de fichier, position courante, pointeurs sur les tampons, indicateurs ...).

  Les flux utilisent une mémoire tampon utilisateur qui est automatiquement vidée quand :

- la mémoire tampon est pleine
- le flux est fermé
- le processus se termine
- en cas d'appel de la fonction de vidage du tampon

Il existe 3 flux définis disponible au démarrage du processus :

- **stdin** : flux correspondant à l'entrée standard
- **stdout** : flux correspondant à la sortie standard
- **stderr** : flux correspondant à la sortie d'erreur standard.

Avantages :

1. Portabilité : le niveau standard est défini dans la bibliothèque standard de la norme ANSI.
2. Gestion bufférisée : rapidité pour les accès séquentiels à un fichier et moins d'appels aux primitives du noyau du système d'exploitation.
3. E/S formatées

La gestion de fichier standard fait appel à la gestion de fichier bas-niveau. Il est déconseillé de mélanger pour un même fichier les deux types de gestion.

Nom de la fonction	description
`fopen`	ouverture ou création
`fclose`	fermeture
`fread`	lecture
`fwrite`	écriture
`fseek`	positionnement
`ftell`	position courante
`fflush`	vidage des tampons
`fprintf`	écriture formatée
`fscanf`	lecture formatée
`getc, fgetc`	lecture d'un caractère
`putc, fputc`	écriture d'un caractère
`fgetw`	lecture d'un entier
`putw`	écriture d'un entier
`fgets`	lecture d'une chaîne de caractères
`fputs`	écriture d'une chaîne de caractères

Figure 11.2: Fonctions de base de la gestion standard des fichiers

Toutes les définitions et déclarations nécessaires à l'utilisation de ces fonctions sont regroupées dans le fichier d'en-têtes `stdio.h`.

Le code exécutable de ces fonctions se trouve dans le fichier archive `/usr/lib/libc.a` qui est automatiquement appelé par l'éditeur de liens.

La manipulation des fichiers se fait par l'intermédiaire d'un pointeur sur une structure de type `FILE` qui est retourné par la fonction d'ouverture.

Ces fonctions positionnent en cas d'erreur la variable `errno` (définie dans `errno.h`) avec le numéro d'erreur. (Attention : le contenu de cette variable n'est valide qu'en cas d'erreur.)

Seules ces fonctions sont traitées dans ce livre. Les fonctions de la gestion de bas niveau sous UNIX, sont traitées dans le livre "Programmation système sous UNIX".

11.1.2 Ouverture d'un fichier (`fopen`)

- **Syntaxe :**

```
#include <stdio.h>

FILE *fopen(const char *nom , const char *mode);
```

- **Arguments :**

`nom` : nom (ou chemin) externe du fichier à ouvrir

`mode` : mode d'ouverture.

- **Description :**

Hormis les flux déjà ouverts, avant de lire ou d'écrire dans un fichier il faut "ouvrir le fichier". Cette opération permet d'associer [1] un flux au fichier désigné par son nom physique `nom`.

- Mode d'ouverture d'un fichier :

- **Remarques :**

  - Contrairement à MSDOS, il n'existe pas sous UNIX de différence entre un *fichier binaire* et un *fichier texte*. Ainsi, il existe sous MSDOS des modes d'ouvertures `rb`, `wb`, `ab`, `r+b`, `w+b`, `a+b` permettant d'ouvrir des fichiers en *mode binaire*.
  - Le nombre maximum de fichiers ouvrables par un processus est défini par la pseudo-constante `FOPEN_MAX`

- **Valeur retournée :**

Si elle aboutit, cette fonction retourne un pointeur sur le flux qui vient d'être ouvert ; sinon, elle retourne `NULL` et positionne la variable `errno`.

[1] si vous en avez le droit

Mode	Description
`r`	Ouverture en lecture seule sur un fichier existant
`w`	Ouverture en écriture seule. Si le fichier existe il est détruit.
`a`	Ouverture pour écriture à la fin du fichier. Création du fichier s'il n'existe pas
`r+`	Ouverture en lecture ou écriture sur un fichier existant.
`w+`	Ouverture en lecture ou écriture. Si le fichier existe il est détruit. Création du fichier s'il n'existe pas
`a+`	Ouverture en lecture ou écriture à la fin du fichier. Création du fichier s'il n'existe pas

Figure 11.3: Mode d'ouverture des fichiers

- **Exemple :** ouverture en création

```
#include <stdio.h>

main(void) {
  FILE *stream;

  if ( (stream = fopen("toto","w")) == NULL) {
      perror("fopen");          /* affichage d'un message d'erreur    */
                                /* correspondant a la valeur de errno */
      exit(1);
  }
  return 0;
}
```

11.1.3 Fermeture d'un fichier (`fclose`)

- **Syntaxe :**

```
#include <stdio.h>

int fclose(FILE *stream);
```

- **Description :** cette fonction ferme le fichier `stream` désigné, après avoir vidé le tampon qui lui est associé.

- **valeur retournée :**

  Elle retourne 0 si elle réussit, dans le cas contraire elle retourne `EOF` (constante de fin de fichier) et positionne la variable `errno`.

- **Remarque :** la fin du processus ferme automatiquement tous les fichiers ouverts.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  FILE *stream;

  fclose(stdin);             /* fermeture de l'entre standard      */
  if ( (stream = fopen("toto","w")) == NULL) {
    perror("fopen");         /* affichage d'un message d'erreur    */
                             /* correspondant a la valeur de errno */
    exit(1);
  }
  fclose(stream);            /* pas de test des erreurs ...        */
  return 0;
}
```

11.1.4 Ecriture dans un fichier (`fwrite`)

- **Syntaxe :**

```
#include <stdio.h>

size_t fwrite(void *ptr, size_t taille, size_t n, FILE *stream);
```

- **Description :**

  Ecriture dans le flux `stream` de `n` objets ayant chacun une longueur de `taille` octets et placés dans une zone pointée par `ptr`.

  `size_t` correspond à un `unsigned int`.

- **Valeur retournée :**

  Elle retourne le nombre d'objets (et non le nombre d'octets) réellement écrits. Le nombre total d'octets écrits est : `n * taille`

- **Exemple 1 :**

```
/************************************************/
/* OUVERTURE ,ECRITURE ET FERMETURE D'UN FICHIER */
/************************************************/

#include <stdio.h>
#include <string.h>   /* strlen() */
#include <stdlib.h>   /* exit() */

main(void) {
  FILE *stream;
  char *msg = "123456789012345";
  int n;
```

```
                                        /* OUVERTURE EN CREATION */
  if ( (stream = fopen("/tmp/toto", "w")) == NULL) {
      perror("fopen");          /* affichage d'un message d'erreur    */
                                /* correspondant a la valeur de errno */
      exit(1);
  }
  n = strlen(msg);
                                        /* ECRITURE DANS LE FICHIER */
  if ( fwrite(msg, sizeof(char), n, stream) != n ) {
                                     /* ecriture dans le flux stderr */
     fwrite("Erreur a l'ecriture\n", 20, 1, stderr);
     exit(2);
  }
                               /* FERMETURE DU FLUX ET VIDAGE DU TAMPON */
  fclose(stream);              /* pas de test des erreurs ...           */
  return 0;
}
```

- **Exemple 2 :**
  - Ecriture d'un entier :

    ```
    int i=10;

    fwrite( &i, sizeof(int), 1, stream);
    ```

  - Ecriture d'un tableau d'entiers :

    ```
    int tab[100];

    fwrite( tab, sizeof(tab[0]), 100, stream);
    ```

  - Ecriture d'une structure et d'un tableau de structures :

    ```
    stuct s_fiche { char nom[20];
                    int  numero;
                  } fiche, tabfiche[100];

    fwrite( &fiche, sizeof(struct s_fiche), 1, stream);
    fwrite( tabfiche, sizeof(struct s_fiche), 100, stream);
    ```

11.1.5 Lecture dans un fichier (`fread`)

- **Syntaxe :**

```
#include <stdio.h>

size_t fread(void *ptr, size_t taille, size_t n, FILE *stream);
```

- **Description :**

  Lecture dans le flux `stream` de `n` objets ayant chacun une longueur de `taille` octets et rangement de ces éléments dans la zone pointée par `ptr`.

- **Valeur retournée :**

  Cette fonction retourne le nombre d'objets réellement lus, qui peut être inférieur au nombre `n`. Elle retourne la valeur `0` si la fin du fichier est rencontrée ou s'il y a une erreur de lecture.

- **Exemple :**

```
/* lecture du fichier cree par l'exemple precedent */

#include <stdio.h>
#include <stdlib.h>   /* exit() */

main(void) {
  FILE *stream;
  char buf[11];
  int nlec;

  /* ouverture en lecture du fichier cree par l'exemple precedent */
  if ( (stream = fopen("/tmp/toto", "r")) == NULL) {
      perror("fopen");         /* affichage d'un message d'erreur    */
                               /* correspondant a la valeur de errno */
      exit(1);
  }
  do {
    /* lecture de 10 caracteres dans le fichier */
    nlec = fread(buf, sizeof(char), 10, stream);
    if (nlec != 0) {  /* affichage des nlec caracteres lus */
       fwrite(buf, sizeof(char), nlec, stdout);
       putchar('\n');
    }
  } while ( nlec == 10 );
  fclose(stream);
  return 0;
}
/*-- resultat de l'execution ----------------------
1234567890
12345
-------------------------------------------------*/
```

11.1.6 Ecriture des tampons (`fflush`)

- **Syntaxe :**

```
#include <stdio.h>

int fflush(FILE *stream);
```

- **Description :**

La mémoire tampon associée à un flux est vidée lorsqu'elle est pleine ou à la fermeture du flux. La fonction `fflush` permet d'écrire sur disque tous les tampons associés au flux `stream`.

- **Valeur retournée :**

  Elle retourne `EOF` lorsqu'une erreur est détectée et positionne la variable `errno`.

- **Remarque :** si `stream` est un pointeur `NULL`, `fflush` vide sur disque les tampons de tous les flux ouverts.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  FILE *stream = fopen("/tmp/essai", "w+");

  fflush( stdin );  /* vidage des tampons associes au flux stdin */
  fflush( stream );
  return 0;
}
```

11.1.7 Lecture d'un caractère (`fgetc`)

- **Syntaxe :**

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);   /* macro identique a la fonction fgetc */
```

- **Description :**

  Lecture d'un caractère depuis le pointeur courant du flux `stream`.

- **Valeur retournée :**

  Retourne le caractère lu ou retourne `EOF` si la fin du fichier est atteinte. En cas d'erreur de lecture, elle retourne aussi `EOF` et positionne la variable `errno`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>   /* strlen() */

int main(void)  {
  FILE *stream;
  char str[] = "123456789012345";
  char ch;

  stream = fopen("/tmp/essai", "w+");
```

```
  fwrite(str, strlen(str), 1, stream);

  fseek(stream, 0, SEEK_SET);  /* se repositionne au debut du fichier */
  do {
    ch = fgetc(stream); /* lecture caractere par caractere du fichier */
    putchar(ch);        /* et affichage sur stdout du caractere lu    */
  } while (ch != EOF);
  fclose(stream);
  return 0;
}
```

11.1.8 Ecriture d'un caractère (fputc)

- **Syntaxe :**

```
#include <stdio.h>

int fputc(int ch, FILE *stream);
int putc(int ch , FILE *stream);    /* macro identique a fputc */
```

- **Description :**

  Ecriture du caractère `ch` dans le flux `stream`.

- **Valeur retournée :**

  Retourne le caractère écrit ou retourne `EOF` en cas d'erreur.

- **Exemple :**

```
/*********************************************/
/* PROGRAMME : COPY_CAR_A_CAR.C              */
/*                                           */
/* COPIE D'UN FICHIER CARACTERE PAR CARACTERE */
/*********************************************/
#include <stdio.h>

main() {
   FILE *in , *out;
               /* Ouverture du fichier en lecture */
   if ( (in = fopen("ESSAI.C" , "r")) == NULL) {
      perror("fopen en lecture");
      exit(1);
   }
               /* Ouverture du fichier en creation */
   if ( (out = fopen("TEMPO.$$$" , "w")) == NULL) {
      perror("fopen en creation");
      exit(1);
   }
   while ( ! feof(in) )
```

```
      putc( getc(in) , out);
   fclose(in);
   fclose(out);
}
```

11.1.9 Lecture d'une chaîne (fgets)

- **Syntaxe :**

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
```

- **Description :**

  Lecture de `n` caractères du flux `stream` et place le résultat à l'endroit pointé par `s`.

  La lecture de la chaîne s'arrête si la fin du fichier est atteinte, ou si un caractère `'\n'` est lu, ou si `n - 1` caractères ont été lus.

- **Valeur retournée :**

  Elle retourne la chaîne et `NULL` en cas de fin de fichier ou d'erreur.

- **Remarques :**

  - Le caractère d'interligne `'\n'` (s'il est lu) est conservé dans la chaîne.
  - Un caractère nul (`'\0'`) est placé en fin de la chaîne pointée par `s`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>    /* strlen() */
#include <stdlib.h>    /* exit() */

main(void) {
    FILE *stream;
    char buf[80];

    if ( (stream = fopen("/tmp/essai", "r")) == NULL ) {
      perror("fopen");
      exit(1);
    }

    /* lecture d'une chaine */
    fgets(buf, sizeof(buf), stream);
    printf("%s", buf);
    fclose(stream);
    return 0;
}
```

11.1.10 Ecriture d'une chaîne (`fputs`)

- **Syntaxe :**

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
```

- **Description :**

  Ecriture de la chaîne pointée par `s` dans le flux `stream`.

- **Valeur retournée :**

  S'il n'y a pas d'erreur, cette fonction retourne le dernier caractère écrit ; sinon, c'est la valeur `EOF` qui est retournée.

- **Exemple :**

```
#include <stdio.h>

main(void) {
    fputs("Hello world\n", stdout);
    return 0;
}
```

11.1.11 Ecriture formatée dans un flux (`fprintf`)

- **Syntaxe :**

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format , argument ... );
```

- **Description :**

  Ecriture formatée dans le fichier `stream`.

  `argument ...` est une liste d'expressions dont les valeurs seront converties suivant les spécifications de la chaîne `format` avant d'être écrites dans le flux `stream`.

  Cette fonction utilise les mêmes spécifications de format que `printf`.

- **Valeur retournée :**

  `fprintf` retourne le nombre d'octets écrits ou `EOF` en cas d'erreur.

- **Exemple :**

```
#include <stdio.h>

main(void) {
    FILE *stream;
    int i = 1234;
    char c = 'C';
    float f = 1.234;

    stream = fopen("/tmp/essai", "w");
    fprintf(stream, "%d %c %f", i, c, f);
    fclose(stream);
    return 0;
}
```

11.1.12 Lecture formatée dans un flux (`fscanf`)

- **Syntaxe :**

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format , pointeur ...);
```

- **Description :** Lecture formatée dans le flux `stream`.

  `pointeur ...` est une liste d'expressions de type pointeur sur des objets qui recevront les valeurs converties suivant les spécifications de la chaîne `format`.

- **Valeur retournée :**

  Cette fonction retourne le nombre de champs d'entrée correctement lus, convertis et mémorisés ou `EOF` en cas d'erreur.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  FILE *stream;
  int i;
  char c;
  float f;

  /* lecture du fichier cree par l'exemple precedent */
  stream = fopen("/tmp/essai", "r");
  if (fscanf(stream, "%d %c %f", &i, &c, &f) == 3)
    printf("Lecture reussie: %d %c %f\n", i, c, f);
  fclose(stream);
  return 0;
}
```

11.1.13 Positionnement du pointeur de fichier (`fseek`)

- **Syntaxe :**

```
#include <stdio.h>

int fseek(FILE *stream , long offset , int methode);
```

- **Description :**

Par défaut, les lectures/écritures s'effectuent à partir de la position courante du pointeur de fichier. Ce dernier est incrémenté automatiquement du nombre de caractères lus ou écrits après chaque opération de lecture/écriture. (accès séquentiel)

Il est possible de modifier la position courante par la fonction `fseek`, permettant ainsi l'accès direct à une information.

`fseek` positionne le pointeur de fichier du flux `stream` pour la prochaine opération de lecture/écriture.

- **Arguments :**

`methode` :

`SEEK_SET (0)` : Positionnement à `offset` octet(s) du début du fichier.

`SEEK_CUR (1)` : Positionnement à la position courante + `offset` octet(s).

`SEEK_END (2)` : Positionnement à la fin du fichier + `offset` octet(s). [2]

- **Valeur retournée :**

Cette fonction retourne 0 si le pointeur a pu être déplacé et une valeur non nulle sinon.

- **Remarques :**

  - L'appel : `fseek(stream,0L,SEEK_SET);` revient à se positionner au début du fichier, il est équivalent à l'instruction `rewind(stream);`
  - Le positionnement dans un flux ouvert en mode `append` ne sert à rien, car il y a positionnement à la fin du fichier avant chaque écriture .

- **Exemple :**

```
#include <stdio.h>
#include <string.h>   /* strlen() */

int main(void)  {
  FILE *stream;
  char str[] = "123456789012345";
  char ch;
```

[2] `offset` peut-être négatif !

```
  stream = fopen("/tmp/essai", "w+");
  fwrite(str, strlen(str), 1, stream);

  /* se repositionne au debut du fichier */
  fseek(stream, 0, SEEK_SET);

  do {
    ch = fgetc(stream); /* lecture caractere par caractere du fichier */
    putch(ch);          /* et affichage sur stdout du caractere lu    */
  } while (ch != EOF);
  fclose(stream);
  return 0;
}
```

11.1.14 Position courante du pointeur de fichier (`ftell`)

- **Syntaxe :**

```
#include <stdio.h>

long ftell(FILE *stream);
```

- **Description :**

Cette fonction retourne la position courante du pointeur de fichier ou la valeur `-1L` quand il y a une erreur.

- **Exemple :**

```
/************************************************/
/* FONCTION QUI COMPTE LE NOMBRE DE LIGNE D'UN    */
/* FICHIER TEXTE A PARTIR DE LA POSITION COURANTE */
/************************************************/

int nb_ligne(f)
FILE *f;
{  int nl = 0;
   long position;

   position = ftell(f);
   while( fgets(ligne , 255 , f) )
      ++nl;
   fseek(f,position,0);    /* retour a la position initiale */
   return nl;
}
```

11.1.15 Tester la fin de fichier (feof)

- **Syntaxe :**

```
#include <stdio.h>

int feof(FILE *stream);
```

- **Description :**

  Cette fonction retourne une valeur non nulle si la fin de fichier désigné par `stream` est atteinte.

  Elle permet donc de distinguer une erreur de lecture ou d'écriture d'une fin de fichier.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  FILE *stream;
  char ch;

  stream = fopen("/tmp/essai", "r");
  ch = fgetc(stream);   /* lecture d'1 caractere dans le fichier */
  if (feof(stream))     /* teste la fin du fichier */
    printf("fin du fichier atteinte\n");
  fclose(stream);
  return 0;
}
```

11.1.16 Gestion des erreurs (ferror et clearerr)

- **Syntaxe :**

```
#include <stdio.h>

int ferror(FILE *stream);
int clearerr(FILE *stream);
```

- **Description :**

  La fonction `ferror` retourne une valeur non nulle après une erreur sur le fichier `stream`. L'erreur reste positionnée jusqu'à l'appel de la fonction `clearerr` ou à la fermeture du fichier.

- **Exemple :**

```
#include <stdio.h>

main(void) {
```

```
  FILE *stream = fopen("/tmp/essai", "w");

  (void) getc(stream);   /* erreur: flux ouvert en ecriture seule */

  if (ferror(stream)) {     /* teste si une erreur c'est produite */
    fprintf(stderr, "Erreur sur le fichier\n");
    clearerr(stream);           /* RAZ de l'erreur et du flag EOF */
  }
  fclose(stream);
  return 0;
}
```

11.1.17 Fonctions diverses (`remove`, `rename`)

- **Syntaxe :**

```
#include <stdio.h>

int remove(const char *nom);
int rename(const char *anciennom , const char *nouveaunom);
```

- **Description :**

  La fonction **remove** supprime le fichier désigné. Elle retourne 0 si l'opération a réussi et une valeur non nulle si elle a échoué.

  La fonction **rename** modifie le nom d'un fichier. `anciennom` est remplacé par `nouveaunom`.

- **Valeur retournée :**

  Elles retournent 0 si l'opération a réussi et une valeur non nulle si elle a échoué.

- **Exemple :**

```
#include <stdio.h>

main(void) {
  char nf[80];

  printf("Fichier a effacer: ");  gets(nf);
  if (remove(nf) == 0)
    printf("Effacement de %s\n", nf);
  else
    perror("remove");
  return 0;
}
```

- **Exemple :**

```
#include <stdio.h>

main(int argc, char *argv[]) {

  if ( argc != 3)
    fprintf(stderr,"Usage: %s ancien_nom nouveau_nom\n", argv[0]);
  else
    if (rename(argv[1], argv[2]) != 0)
      perror("rename");
  return 0;
}
```

11.1.18 Redéfinir un flux (freopen)

- **Syntaxe :**

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

- **Description :**

  La fonction `freopen` permet de changer le fichier associé à un flux déjà ouvert.

  `freopen` commence par fermer le flux `stream`. Elle effectue ensuite l'ouverture d'un nouveau flux associé au fichier de nom `filename` et selon le mode décrit par le paramètre `mode`. Si l'ouverture réussit, le nouveau flux est placé dans le descripteur `stream`.

  Le flux original `stream` est refermé, même si l'ouverture du nouveau fichier échoue.

- **Valeur retournée :**

  En cas de succès `freopen` retourne le flux fourni en argument. Si erreur, la fonction retourne `NULL`.

- **Exemple 1 :** redirection de la sortie standard

```
#include <stdio.h>

main(void) {

    /* redirection de stdout dans un fichier */
    if ( freopen("tempo.log", "w", stdout) == NULL)
       fprintf(stderr, "erreur a la redirection de stdout\n");

    /* Cet affichage va dans le fichier  tempo.log */
    printf("Cet affichage va dans le fichier\n");

    /* fermeture du flux stdout */
```

```
    fclose(stdout);

    return 0;
}
```

- **Exemple 2 :** écriture forcée sur l'écran du terminal

```
/* freopen.c */
#include <stdio.h>

main(void) {

  printf("Coucou...\n");
  /* redirection de stdout vers /dev/tty */
  if ( freopen("/dev/tty", "w", stdout) == NULL)
    fprintf(stderr, "erreur a la redirection de stdout\n");

  /* Cet affichage va sur l'ecran du terminal associe au processus */
  printf("Cet affichage va obligatoirement a l'ecran\n");
  printf("meme en cas de redirection de stdout\n");

  fclose(stdout);
  return 0;
}
```

Exemple d'exécution de ce programme :

```
apollo32:[~]> freopen
Coucou...
Cet affichage va obligatoirement a l'ecran
meme en cas de redirection de stdout

apollo32:[~]> freopen > /dev/null
Cet affichage va obligatoirement a l'ecran
meme en cas de redirection de stdout
```

Dans la deuxième exécution du programme, l'affichage de `coucou` est redirigé vers le fichier `/dev/null`.

11.2 Les fonctions mathématiques

Le fichier de déclaration des fonctions mathématiques est le fichier d'en-têtes `math.h`. Le code exécutable de ces fonctions est dans la librairie `/usr/lib/libm.a` qu'il faut préciser au linker.

11.2.1 Valeur absolue d'un réel (fabs)

- **Syntaxe :**

```
#include <math.h>

double fabs(double x);
```

- **Description :** retourne la valeur absolue de `x`.

- **Exemple :**

```
#include <stdio.h>
#include <math.h>

main(void) {
    double nbre = -1234.0;

    printf("Valeur absolue de %lf =  %lf\n", nbre, fabs(nbre) );
    return 0;
}
/*-- resultat de l'execution ---------------------------------
Valeur absolue de -1234.000000 =  1234.000000
------------------------------------------------------------*/
```

11.2.2 Valeur absolue d'un entier (abs, labs)

- **Syntaxe :**

```
#include <stdlib.h>

int abs(int x);
long labs(long x);
```

- **Description :**

  - La macro `abs` renvoie la valeur absolue d'un entier.
  - `labs` retourne la valeur absolue d'un entier long.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>

main(void) {
  int n1  = -1234;
  long n2 = -12345678L;

  printf("La valeur absolue de l'entier %d est %d\n", n1, abs(n1));
```

```
  printf("La valeur absolue de l'entier long %ld est %ld\n", n2, labs(n2));
  return 0;
}
/*-- resultat de l'execution ---------------------------------
La valeur absolue de l'entier -1234 est 1234
La valeur absolue de l'entier long -12345678 est 12345678
------------------------------------------------------------*/
```

11.2.3 Fonctions trigonométriques

Syntaxe	Description
`double acos(double x);`	Arc cosinus de `x`. Retourne entre 0 et π
`double asin(double x);`	Arc sinus de `x`. Retourne entre $-\pi/2$ et $\pi/2$
`double atan(double x);`	Arc tangente de `x`. Retourne entre $-\pi/2$ et $\pi/2$
`double atan2(double y, double x);`	Arc tangente de `y/x`. Retourne entre $-\pi$ et π
`double cos(double x);`	Cosinus de `x` en radian.
`double sin(double x);`	Sinus de `x` en radian.

11.2.4 Fonctions hyperboliques

Syntaxe	Description
`double cosh(double x);`	cosinus hyperbolique de `x`
`double sinh(double x);`	sinus hyperbolique
`double tanh(double x);`	tangente hyperbolique

11.2.5 Fonctions exponentielle et puissance

Syntaxe	Description
`double exp(double x);`	exponentielle e^x
`double log(double x);`	logarithme népérien (ln)
`double log10(double x);`	logarithme décimal
`double pow(double x, double y);`	puissance x^y
`double sqrt(double x);`	racine carrée
`double ldexp(double x, int *n);`	Retourne $x.2^n$
`double frexp(double val, int *expo);`	représentation flottante normalisée Calcule `x` et `n`, pour obtenir $val = x.2^n$ retourne la mantisse `x` et place l'exposant `n` à l'adresse pointée par `expo`

11.2.6 Arrondi (`ceil`, `floor`)

- Arrondit pas excès :

```
#include <math.h>

double ceil(double x);
```

- Arrondit pas défaut :

```
#include <math.h>

double floor(double x);
```

- **Exemple :**

```
#include <stdio.h>
#include <math.h>

main(void) {
   double nbre = 1234.56;

   printf("ceil de  %lf = %lf\n", nbre, ceil(nbre) );
   printf("floor de %lf = %lf\n", nbre, floor(nbre) );
   return 0;
}
/*-- resultat de l'execution ------------------------------------
ceil de  1234.560000 = 1235.000000
floor de 1234.560000 = 1234.000000
-----------------------------------------------------------------*/
```

11.2.7 Modulo et décomposition (`fmod`, `modf`)

- Modulo réel :

```
#include <math.h>

double fmod(double x, double y);
```

- Décomposition partie entière/partie décimale :

```
#include <math.h>

double modf(double x, double *part_entier);
```

Cette fonction décompose le nombre `x` en sa partie entière et sa partie décimale.

Elle place la partie entière à l'adresse pointée par `part_entier` et retourne la partie décimale.

- **Exemple :**

```
#include <stdio.h>
#include <math.h>

main(void) {
  double n1 = 1234.5678, pd, pe;

  printf("%lf modulo 10.0 = %lf\n", n1, fmod(n1, 10.0));
  pd = modf( n1, &pe);
  printf("%lf   %lf\n", pe, pd);

  return 0;
}
/*-- resultat de l'execution -------------------------
1234.567800 modulo 10.0 = 4.567800
1234.000000   0.567800
-----------------------------------------------------*/
```

11.2.8 Constantes mathématiques

Le fichier `math.h` contient diverses constantes souvent utilisées :

M_E	2.7182818284590452354	e
M_LOG2E	1.4426950408889634074	$1/\ln 2$
M_LOG10E	0.43429448190325182765	$\log 2$
M_LN2	0.69314718055994530942	$\ln 2$
M_LN10	2.30258509299404568402	$\ln 10$
M_PI	3.14159265358979323846	π
M_PI_2	1.57079632679489661923	$\pi/2$
M_PI_4	0.78539816339744830962	$\pi/4$
M_1_PI	0.31830988618379067154	$1/\pi$
M_2_PI	0.63661977236758134308	$2/\pi$
M_2_SQRTPI	1.12837916709551257390	$2/\sqrt{\pi}$
M_SQRT2	1.41421356237309504880	$\sqrt{2}$
M_SQRT1_2	0.70710678118654752440	$\sqrt{1/2}$
MAXFLOAT	3.40282346638528860e+38	

11.2.9 Gestion des erreurs mathématiques (`matherr`)

- **Syntaxe :**

```
#include <math.h>

int matherr(struct exception *e);
```

- **Description :**

  La fonction `matherr` est appelée quand une erreur survient lors de l'exécution d'une fonction de la librairie mathématique.

  `matherr` permet de surdéfinir une fonction de gestion des erreurs mathématiques conçue par l'utilisateur pour gérer les erreurs de domaines de définition et de débordement générées par les fonctions mathématiques.

- **Remarques :**

  - Elle ne permet pas de résoudre les exceptions de l'arithmétique flottante telles que la division par zéro [3]
  - Le comportement standard `matherr` sous UNIX n'est pas compatible avec la norme ANSI.

- **Valeur retournée :**

  - La fonction `matherr` surdéfinie doit renvoyer une valeur non nulle si elle réussit à résoudre l'erreur et 0 sinon.
  - Si la fonction `matherr` n'est pas surdéfinie, la valeur renvoyée est 1 en cas d'erreur `UNDERFLOW` ou `TLOSS`, sinon elle retourne 0.

  Quand `matherr` renvoie une valeur non nulle, aucun message d'erreur n'est affiché et la variable globale `errno` [4] n'est pas modifiée.

- **Argument :**

  Le type `struct exception` :

```
struct exception {
   int     type;
   char   *name;
   double  arg1, arg2, retval;
};
```

  - `type` : type d'erreur mathématique survenue :

`DOMAIN`	Argument hors domaine
`SING`	L'argument aurait donné une singularité
`OVERFLOW`	L'argument aurait donné un résultat trop grand
`UNDERFLOW`	L'argument aurait donné un résultat trop petit
`TLOSS`	L'argument aurait donné une perte totale des chiffres
`PLOSS`	L'argument aurait donné une perte partielle des chiffres

  - `name` : contient le nom de la fonction mathématique liée à l'erreur.
  - `arg1` : premier argument de la fonction mathématique.

[3] Pour la gestion de ce type d'erreur, voir la primitive `signal` de l'ouvrage "Programmation système sous UNIX"

[4] Déclarée dans `errno.h`

- `arg2` : deuxième argument.
- `retval` : valeur renvoyée à l'appelant.

- **Exemple :**

```
#include <stdio.h>
#include <math.h>
#include <string.h>

int matherr (struct exception *x) {
   switch ( x->type ) {
     case DOMAIN :
            if (strcmp(x->name, "sqrt")==0) {
               x->retval = sqrt(-x->arg1);
               return 1;  /* n'execute pas la procedure par defaut */
            }
            break;
     default:
            fprintf(stderr, "ERREUR MATH: %s(%g) retourne %g\n",
                            x->name, x->arg1, x->retval);
   }
   return 0;  /* execute la procedure par default */
}

main(void) {
   double x = -2.0, y;

   y = sqrt(x);      /* DOMAIN */
   printf("Racine carree de %lf = %lf\n", x, y);

   y = exp(1000);    /* OVERFLOW */
   y = exp(-1000);   /* UNDERFLOW */
   y = sin(1e100);    /* TLOSS */
   return 0;
}
/*-- Resultat de l'execution ---------------------------
Racine carree de -2.000000 = 1.414214
ERREUR MATH: exp(1000) retourne 1.79769e+308
exp: OVERFLOW error
ERREUR MATH: exp(-1000) retourne 0
exp: UNDERFLOW error
ERREUR MATH: sin(1e+100) retourne +NAN
sin: TLOSS error
-----------------------------------------------------------*/
```

11.3 Taille des type entiers (`limits.h`)

Le fichier `limits.h` définit des pseudo-constantes pour les tailles des types entiers.

`CHAR_BIT`	Nombre de bits du type `char`
`CHAR_MIN`	Valeur maximale du type `char`
`CHAR_MAX`	Valeur minimale du type `char`
`SHRT_MIN`	Valeur minimale du type `short`
`SHRT_MAX`	Valeur maximale du type `short`
`INT_MIN`	Valeur minimale du type `int`
`INT_MAX`	Valeur maximale du type `int`
`LONG_MIN`	Valeur minimale du type `long`
`LONG_MAX`	Valeur maximale du type `long`
`UCHAR_MAX`	Valeur minimale du type `unsigned char`
`USHRT_MAX`	Valeur maximale du type `unsigned short`
`UINT_MAX`	Valeur maximale du type `unsigned int`
`ULONG_MAX`	Valeur maximale du type `unsigned long`

11.4 Limites des type réels (`float.h`)

Le fichier `float.h` définit des pseudo-constantes qui caractérisent les types réels.

`FLT_DIG`	nombre de chiffres significatifs d'un `float`
`DBL_DIG`	nombre de chiffres significatifs d'un `double`
`LDBL_DIG`	nombre de chiffres significatifs d'un `long double`
`FLT_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`DBL_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`LDBL_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`FLT_MAX`	le plus grand nombre du type `float`
`DBL_MAX`	le plus grand nombre du type `double`
`LDBL_MAX`	le plus grand nombre du type `long double`
`FLT_MIN`	le plus petit nombre du type `float`
`DBL_MIN`	le plus petit nombre du type `double`
`LDBL_MIN`	le plus petit nombre du type `long double`

11.5 Traitement de chaînes de caractères

Toutes les fonctions de traitement de chaînes de caractères s'appliquent à des chaînes terminées par un caractère nul (`'\0'`).

Le prototype de toutes ces fonctions est déclaré dans le fichier en-tête `string.h`.
Le type `size_t` est défini comme un `unsigned int`.

11.5.1 Copie de chaîne (`strcpy`)

- **Syntaxe :**

```
#include <string.h>

char *strcpy(char *dest, const char *srce);
```

- **Description :** copie de la chaîne `srce` dans la chaîne `dest`, y compris le caractère nul de fin de chaîne.

  La chaîne `dest` doit être auparavant allouée statiquement ou dynamiquement.

- **Valeur retournée :** l'adresse `dest`

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char str[10];
    char *ptr = "ABCDEFGHIJ";

    strcpy(str, ptr);
    printf("%s\n", str);
    return 0;
}
/* resultat de l'execution ----------------------------
ABCDEFGHIJ
------------------------------------------------------*/
```

11.5.2 Copie partielle de chaîne (`strncpy`)

- **Syntaxe :**

```
#include <string.h>

char *strncpy(char *dest, const char *srce, size_t maxlen);
```

- **Description :** `strncpy` copie jusqu'à `maxlen` caractères de la chaîne `srce` dans la chaîne `dest`.

  Attention : si le caractère nul de fin de chaîne n'est pas copié, il n'est pas ajouté à la fin de la chaîne `dest`.

- **Valeur retournée :** l'adresse `dest`

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char str[10];
  char *ptr = "ABCDEFGHIJ";

  strncpy(str, ptr, 3);
```

```
  str[3] = '\0';
  printf("%s\n", str);
  return 0;
}
/* resultat de l'execution ----------------------------
ABC
-------------------------------------------------------*/
```

11.5.3 Longueur d'une chaîne (`strlen`)

- **Syntaxe :**

```
#include <string.h>

size_t strlen(const char *str);
```

- **Description :** `strlen` permet de calculer la longueur de `str`.
- **Valeur retournée :** le nombre de caractères de la chaîne `str`, le caractère nul de fin de chaîne n'étant pas compté.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char *str = "E.N.A.C.";

   printf("%d\n", strlen(str));
   return 0;
}
/* resultat de l'execution ----------------------------
8
-------------------------------------------------------*/
```

11.5.4 Concaténation de chaîne (`strcat`)

- **Syntaxe :**

```
#include <string.h>

char *strcat(char *dest, const char *srce);
```

- **Description :** `strcat` copie la chaîne `srce` à la fin de la chaîne `dest` (concaténation de chaînes).

  La longueur de la chaîne résultante est `strlen(dest) + strlen(srce)`.

  Attention : la taille de la chaîne destination doit être suffisante pour recevoir tous les caractères de la chaîne concaténée.

- **Valeur retournée :** `strcat` renvoie l'adresse `dest`,
- **Exemple :**

```
#include <string.h>
#include <stdio.h>

int main(void) {
    char m1[60] = "Ecole Nationale";
    char *m2 = " de l'Aviation Civile";

    strcat(m1, m2);
    printf("%s\n", m1);
    return 0;
}
/* resultat de l'execution ----------------------------
Ecole Nationale de l'Aviation Civile
-------------------------------------------------------*/
```

11.5.5 Concaténation partielle de chaîne (`strncat`)

- **Syntaxe :**

```
#include <string.h>

char *strncat(char *dest, const char *srce, size_t maxlen);
```

- **Description :** `strncat` recopie au plus `maxlen` caractères de la chaîne `srce` à la fin de la chaîne `dest`, puis ajoute un caractère `'\0'`.

  La longueur maximale de la chaîne résultante est `strlen(dest) + maxlen` et `dest` doit être de taille suffisante pour la recevoir.

- **Valeur retournée :** `dest`
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char dest[9] = "E.N.", *srce = "A.C....";

    strncat(dest, srce, 4);
    printf("%s\n", dest);
    return 0;
}
/* resultat de l'execution ----------------------------
E.N.A.C.
-------------------------------------------------------*/
```

11.5.6 Comparaison de chaînes (strcmp)

- **Syntaxe :**

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

- **Description :** `strcmp` effectue une comparaison des deux chaînes `s1` et `s2`.

  La comparaison débute avec le premier caractère de chaque chaîne et continue avec les suivants, jusqu'à ce que des caractères de même rang soient différents ou que l'une des chaînes soit terminée.

- **Valeur retournée :** `strcmp` retourne :

  - une valeur négative si `s1 < s2`
  - une valeur nulle si les deux chaînes sont égales
  - une valeur positive si `s1 > s2`

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "ABCDE", *str2 = "abc";
  int v;

  v = strcmp(str1, str2);
  if (v > 0)
    printf("%s > %s\n", str1, str2);
  else
    if ( v == 0 )
      printf("%s == %s\n", str1, str2);
    else
      printf("%s < %s\n", str1, str2);
  return 0;
}
/* resultat de l'execution ----------------------------
ABCDE < abc
-------------------------------------------------------*/
```

11.5.7 Comparaison partielle de chaînes (strncmp)

- **Syntaxe :**

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t maxlen);
```

- **Description :** cette fonction compare `s1` et `s2`, en cherchant au plus `maxlen` caractères.

  La comparaison débute avec le premier caractère de chaque chaîne et continue avec les caractères suivants, jusqu'à ce que des caractères de même rang soient différents ou que `maxlen` soit atteint ou que l'une des chaînes soit terminée.

- **Valeur retournée :** `strncmp` retourne :

  - une valeur négative si `s1 < s2`
  - une valeur nulle si les deux chaînes sont égales
  - une valeur positive si `s1 > s2`

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "ABCDEF", *str2 = "ABCdef";
  int v;

  v = strncmp(str1, str2, 4);
  if (v > 0)
    printf("%s > %s  ", str1, str2);
  else
    if ( v == 0 )
      printf("%s == %s  ", str1, str2);
    else
      printf("%s < %s  ", str1, str2);
  printf("(sur les 4 premiers caracteres)\n");
  return 0;
}
/* resultat de l'execution ----------------------------
ABCDE < abc
-------------------------------------------------------*/
```

11.5.8 Comparaison de chaînes (`strcasecmp`, `strncasecmp`)

- **Syntaxe :**

```
#include <string.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t maxlen);
```

- **Description :**

  Ces fonctions sont identiques à `strcmp` et `strncmp` respectivement, mais la comparaison se fait sans distinguer les majuscules des minuscules (appel à la macro `_tolower` avant la comparaison).

11.5.9 Recherche de caractère (strchr, index)

- **Syntaxe :**

```
#include <string.h>

char *strchr(const char *s, int c);
char *index (const char *s, int c);
```

- **Description :** `strchr` recherche la première occurrence du caractère `c` dans la chaîne de caractères `s`.

  Le caractère nul de fin de chaîne est considéré comme en faisant partie; ainsi l'expression `strchr(str, 0)` renvoie un pointeur vers celui ci.

  La fonction `index` est identique à `strchr` et est disponible pour des raisons de portabilité des applications BSD.

- **Valeur retournée :**

  - En cas de succès, `strchr` retourne un pointeur sur la première occurrence du caractère `c`.
  - Si `c` ne figure pas dans la chaîne, `strchr` retourne `NULL`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str = "E.N.A.C.", c = 'A', *ptr;

  if ( (ptr = strchr(str, c)) != NULL )
  if (ptr)
    printf("Le caractere %c est a la position %d\n", c, ptr-str);
  else
    printf("Le caractere %c n'est pas trouve dans %s\n", c, str);
  return 0;
}
/* resultat de l'execution ----------------------------
Le caractere A est a la position 4
-------------------------------------------------------*/
```

11.5.10 Recherche à rebours de caractère (strrchr, rindex)

- **Syntaxe :**

```
#include <string.h>

char *strrchr(const char *s, int c);
char *rindex (const char *s, int c);
```

- **Description :** cherche dans une chaîne de caractères `s` la dernière occurrence du caractère `c`.

  Le caractère nul de fin de chaîne est considéré comme faisant partie de celle ci.

  La fonction `rindex` est identique à `strrchr` et est disponible pour des raisons de portabilité des applications BSD.

- **Valeur retournée :**

  - En cas de succès, `strrchr` renvoie un pointeur sur la dernière occurrence du caractère `c`.
  - Si `c` ne figure pas dans `s`, `strrchr` renvoie le pointeur `NULL`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str="E.N.A.C.", c = '.', *ptr;

  if ( (ptr = strrchr(str, c)) != NULL)
    printf("Le caractere %c est en position %d\n", c, ptr-str);
  else
    printf("Pas trouve\n");
  return 0;
}
/* resultat de l'execution ----------------------------
Le caractere . est en position 7
-------------------------------------------------------*/
```

11.5.11 Recherche d'un caractère d'un ensemble (`strpbrk`)

- **Syntaxe :**

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

- **Description :** `strpbrk` examine la chaîne `s1` à la recherche de la première occurrence d'un caractère figurant dans la chaîne `s2`.

- **Valeur retournée :**

  - En cas de succès, `strpbrk` retourne un pointeur sur la première occurrence dans `s1` d'un caractère de `s2`.
  - Si aucun caractère de `s2` ne figure dans `s1`, le pointeur `NULL` est retourné.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "ECOLE NATIONALE DE L'AVIATION CIVILE";
  char *str2 = "DCL";
  char *ptr;

  if ( (ptr = strpbrk(str1, str2)) != NULL )
    printf("Trouve : %c en position %d\n", *ptr, ptr-str1);
  else
    printf("Pas trouve\n");
  return 0;
}
/* resultat de l'execution ----------------------------
Trouve : C en position 1
-------------------------------------------------------*/
```

11.5.12 Duplication d'une chaîne (`strdup`)

- **Syntaxe :**

```
#include <string.h>

char *strdup(const char *s);
```

- **Description :** `strdup` fait un double de la chaîne `s` en allouant de la place par un appel à la fonction `malloc`.

  L'espace alloué a pour taille `(strlen(s) + 1)` octets.

  Quand la mémoire allouée n'est plus nécessaire, l'utilisateur devra la libérer par un appel à la fonction `free`.

- **Valeur retournée :**

  - Si succès, `strdup` renvoie un pointeur sur l'emplacement de mémoire contenant le double de la chaîne `s`.
  - Si erreur, `strdup` renvoie le pointeur `NULL` (l'espace nécessaire pour le double de la chaîne n'a pu être alloué).

- **Exemple :**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>  /* free() */

int main(void) {
    char *str1 = "ABCDEFGHIJ", *str2;
```

```
    str2 = strdup(str1);
    printf("%s\n", str2);
    free(str2);
    return 0;
 }
/* resultat de l'execution ----------------------------
ABCDEFGHIJ
-------------------------------------------------------*/
```

11.5.13 Recherche d'une sous-chaîne (strstr)

- **Syntaxe :**

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

- **Description :** `strstr` recherche la première occurrence de la chaîne `s2` dans la chaîne `s1`.
- **Valeur retournée :**
  - En cas de succès, `strstr` retourne un pointeur sur l'élément de `s1` où commence `s2`.
  - Si `s2` n'est pas dans `s1`, `strstr` renvoie `NULL`.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "Ecole Nationale de l'Aviation Civile";
  char *str2 = "at", *ptr;

  ptr = strstr(str1, str2);
  printf("%s\n", ptr);
  return 0;
}
/* resultat de l'execution ----------------------------
ationale de l'Aviation Civile
-------------------------------------------------------*/
```

11.5.14 Recherche à rebours d'une sous-chaîne (strrstr)

- **Syntaxe :**

```
#include <string.h>

char *strrstr(const char *s1, const char *s2);
```

- **Description :** `strstr` recherche la dernière occurrence de la chaîne `s2` dans la chaîne `s1`.
- **Valeur retournée :**
  - En cas de succès, `strstr` retourne un pointeur sur l'élément de `s1` où commence `s2`.
  - Si `s2` n'est pas dans `s1`, `strstr` renvoie `NULL`.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "Ecole Nationale de l'Aviation Civile";
  char *str2 = "at", *ptr;

  ptr = strrstr(str1, str2);
  printf("%s\n", ptr);
  return 0;
}
/* resultat de l'execution ----------------------------
ation Civile
------------------------------------------------------*/
```

11.5.15 Recherche d'une sous-chaîne (`strspn`, `strcspn`)

- **Syntaxe :**

```
#include <string.h>

size_t strspn (const char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
```

- **Description :**
  - `strspn` retourne le nombre de caractères du début de la chaîne `s1` formée uniquement par des caractères figurant dans `s2`.
  - `strcspn` retourne le nombre de caractères du début de la chaîne `s1` formée uniquement par des caractères ne figurant pas dans `s2`.
- **Valeur retournée :** la longueur du premier segment trouvé.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char *str1 = "ECOLE NATIONALE DE L'AVIATION CIVILE";
  char *str2 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  printf("%d\n", strspn(str1, str2) );
  return 0;
}
/* resultat de l'execution ----------------------------
5
-------------------------------------------------------*/
```

11.5.16 Recherche des lexèmes dans une chaîne (`strtok`)

- **Syntaxe :**

```
#include <string.h>

char *strtok(char *s1, const char *s2);
```

- **Description :** `strtok` décompose la chaîne `s1` en sous-chaînes appelées lexèmes (tokens). Ces lexèmes sont délimités par des caractères (séparateurs) figurant dans la chaîne `s2`.

  Le premier appel à `strtok` :

  - retourne un pointeur sur le premier lexème de `s1`
  - écrit un caractère nul dans `s1` immédiatement après ce lexème.

  Les autres appels à `strtok` avec `NULL` comme premier argument traitent de la même manière, et jusqu'à épuisement, les autres lexèmes de `s1`.

  La chaîne des séparateurs, s2, peut être différente d'un appel à l'autre.

- **Valeur retournée :**

  - En cas de succès, renvoie un pointeur sur le lexème trouvé dans la chaîne `s1`.
  - Quand il ne reste plus de lexème dans `s1`, `strtok` retourne le pointeur `NULL`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char ligne[] = "ls  |  tee /dev/tty  >  toto";
    char sep[] ="|>";
    char *ptr;

    ptr = strtok(ligne, sep);
    while ( ptr != NULL ) {
      printf("token: %s\n", ptr);
      ptr = strtok(NULL, sep);
    }
    printf("ligne apres traitement: %s\n", ligne);
    return 0;
}
/* resultat de l'execution ----------------------------
token: ls
token:   tee /dev/tty
token:   toto
ligne apres traitement: ls
-------------------------------------------------------*/
```

11.6 Le traitement de caractères (`ctype.h`)

11.6.1 Fonctions de test d'un caractère

Elles retournent une valeur non nulle (vrai) si la valeur de l'argument `car` est conforme à la description de la fonction.

- `int isalnum(int car);` : teste si `car` est aphanumérique
- `int isalpha(int car);` : teste si `car` est alphabétique
- `int isascii(int car);` : teste si `car` est un caractère ASCII
- `int iscntrl(int car);` : teste si `car` est un caractère de contrôle
- `int isdigit(int car);` : teste si `car` est un chiffre
- `int isgraph(int car);` : teste si `car` est affichable (sauf l'espace)
- `int islower(int car);` : teste si `car` est un caractère minuscule
- `int isprint(int car);` : teste si `car` est affichage
- `int ispunct(int car);` : teste si `car` est un caractère de ponctuation
- `int isspace(int car);` : teste si `car` est un caractère d'espacement
- `int isupper(int car);` : teste si `car` est un caractère majuscule
- `int isxdigit(int car);` : teste si `car` est un chiffre hexadécimal.

11.6.2 Fonctions de conversions de caractères

- `int  toascii(int car);` : convertit `car` en ASCII 7 bits
- `int  tolower(int car);` : convertit `car` en minuscule
- `int  toupper(int car);` : convertit `car` en majuscule.
- `int _tolower(int car);` : macro qui convertit `car` en minuscule
- `int _toupper(int car);` : macro qui convertit `car` en majuscule.

Remarque :
Les deux dernières macro-instructions ne testent pas les erreurs.

11.7 Utilitaires généraux (`stdlib.h`)

11.7.1 Conversions de chaînes (`atof`, `atoi`, `atol`)

- **Syntaxe :**

```
#include <stdlib.h>

double atof(const char *s);
int    atoi(const char *s);
long   atol(const char *s);
```

- **Description :** ces fonctions convertissent la chaîne de caractères `s` en une valeur numérique de type `double`, `int` ou `long`.

  Elles s'arrêtent au premier caractère qui n'est pas reconnu et retournent la valeur `0` si la chaîne ne peut pas être convertie dans le type désiré.

- **Exemple :**

```
d = atof("+1234.56 e-7");
```

11.7.2 Conversions d'un entier en chaîne (`ltoa` , `ltostr`)

- **Syntaxe :**

```
#include <stdlib.h>

char *ltostr(long nbre, int base);
char *ultostr(unsigned long nbre, int base);
char *ltoa(long nbre);
char *ultoa(unsigned long nbre);
```

- **Description :** ces fonctions convertissent le nombre `nbre` en une chaîne de caractères.

- **Exemple :**

```
printf("%s\n",ltostr(18L , 2));     /* 10010 */
printf("%s\n",ltoa(123L));          /* 123   */
```

11.7.3 Nombres aléatoires (`rand` et `srand`)

- **Syntaxe :**

```
#include <stdlib.h>

int  rand(void);
void srand(unsigned int seed);
```

- **Description :**

La fonction `rand` retourne un entier positif pseudo-aléatoire entre 0 et 32767 (`RAND_MAX`).

`srand` modifie la suite de nombres pseudo-aléatoires par assignation d'une valeur de départ au générateur de nombres. La valeur de départ par défaut est la valeur `1`.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void) {
   int i;

   srand(time(NULL));
   for (i = 0 ; i < 5 ; i ++ )
       printf("%d  ", rand() );
}
/*-- resultat de l'execution ---------------------------------
6023  20196  13614  9135  28533
------------------------------------------------------------*/
```

Ici le générateur de nombres pseudo-aléatoires est initialisé avec le nombre de secondes écoulées depuis le 1er janvier 1970.

11.7.4 Gestion de processus (exit, abort, system)

- **Syntaxe :**

```
#include <stdlib.h>

void abord(void);
void exit(int status);
int  system(const char *s);
```

- **Description :**
  - la fonction **abort** met fin au processus en cours en cas d'anomalie. Le signal `SIGABRT` est émis.
  - la fonction **exit** met fin au processus en cours et retourne au processus père la valeur de l'argument `status`.

    En général on transmet une valeur nulle pour indiquer une fin normale et une valeur différente en cas d'erreur.
  - la fonction **system** permet le lancement d'une commande shell. Le processus est interrompu jusqu'à la fin de l'exécution de cette commande.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>

main(void) {
    system("clear"); /* effacement de l'ecran */

    return 0;
}
```

11.7.5 Recherche binaire d'un objet dans un tableau (bsearch)

- **Syntaxe :**

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
      size_t size, int (*compar)(const void*, const void*));
```

- **Description :**

  Recherche dichotomique de la clé `*key` dans le tableau `base`. Les objets du tableau `base` doivent être rangés dans l'ordre croissant.

- **Arguments :**
  - `key` : élément à rechercher (clé de recherche)

- `base` : adresse de l'élément 0 du tableau à scruter
- `nel` : nombre d'éléments dans le tableau
- `size` : taille de chaque élément dans le tableau
- `fcmp` : fonction de comparaison définie par le programmeur pour comparer deux éléments.

- **Fonction de comparaison :**

  Pour réaliser une recherche dans le tableau, `bsearch` fait appel à la fonction dont l'adresse est fournie dans `compar` avec comme premier argument la clé de recherche et comme second argument l'élément du tableau à comparer avec la clé puis retourne un entier donnant le résultat de la comparaison :

  - valeur négative : la clé est plus petite que l'élément en cours
  - valeur nulle : la clé et l'élément en cours sont égaux
  - valeur positive : la clé est plus grande que l'élément en cours

- **Valeur retournée :**

  En cas de succès, `bsearch` retourne l'adresse de la première entrée identique à la clé, sinon `bsearch` retourne `NULL`.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>   /* strcmp() */

typedef int (*fptr)(const void *, const void *);

int compare1(const int *n1, const int *n2) {
    return(*n1 - *n2);
}

int compare2(const char **str1, char **str2) {
    return strcmp(*str1, *str2);
}

main(void) {
  int tab1[9] = {123, 234, 345, 456, 567, 678, 789, 890, 901};
  int key1 = 890;
  char *tab2[5] = {"ABCDEF", "EB", "PHM", "DA", "DF"};
  char *key2 = "DA";

  if ( bsearch(&key1, tab1, 9, sizeof(int), (fptr) compare1) != NULL )
     printf("%d est dans tab1\n", key1);
  else
     printf("%d n'est pas dans tab1\n", key1);
```

```
  if ( bsearch(&key2, tab2, 5, sizeof(char *), (fptr) compare2) != NULL )
     printf("%s est dans tab2\n", key2);
  else
     printf("%s n'est pas dans tab2\n", key2);
  return 0;
}
/*-- resultat de l'execution --------------------------------
890 est dans tab1
DA est dans tab2
------------------------------------------------------------*/
```

11.7.6 Tri par algorithme rapide "quicksort" (qsort)

- **Syntaxe :**

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t size,
           int (*compare)(const void *, const void *));
```

- **Description :**

  `qsort` trie dans l'ordre croissant un tableau de `nel` éléments de taille `size`.

- **Arguments :** voir la fonction `bsearch`.

- **Valeur retournée :** aucune.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>    /* strcmp() */

typedef int (*fptr)(const void *, const void *);

int compare1(const int *n1, const int *n2) {
    return(*n1 - *n2);
}

int compare2(const char **str1, const char **str2) {
  return strcmp(*str1, *str2);
}

main(void) {
  int  i;
  int tab1[9] = {4567, 901, 345, 12, 789, 890, 567, 678, 2345};
  char *tab2[] = {"ABCDEF", "EB", "PHM", "AD", "DF"};

  qsort((void *) tab1, 9, sizeof(int), (fptr) compare1);
```

```
  for (i = 0; i < 9; i++)
     printf("%d ", tab1[i]);
  putchar('\n');
  qsort((void *) tab2, 5, sizeof(char *), (fptr) compare2);
  for (i = 0; i < 5; i++)
     printf("%s ", tab2[i]);
  putchar('\n');
  return 0;
}
/*-- resultat de l'execution --------------------------------
12 345 567 678 789 890 901 2345 4567
ABCDEF AD DF EB PHM
-------------------------------------------------------------*/
```

11.8 Allocation dynamique de mémoire

Le tas (heap) sert à l'allocation dynamique de blocs de mémoire de taille variable.

De nombreuses structures de données emploient tout naturellement l'allocation de mémoire dans le tas, comme par exemple les arbres et les listes.

Il ne faut pas hésiter à utiliser ces fonctions d'allocation mémoire, qui présentent l'avantage (par rapport à une allocation statique des objets, tableau par exemple) d'ajuster la taille des objets à l'exécution et de les libérer quand ils deviennent inutiles.

Le seul risque est la fragmentation du tas, par allocation et libération successives. Il n'existe pas en C de mécanisme de "ramasse-miettes" (garbage collector).

11.8.1 Demande d'allocation mémoire (`malloc`)

- **Syntaxe :**

```
#include <stdlib.h>

void *malloc(size_t size);
```

- **Description :**

  `malloc` demande l'allocation d'un bloc de mémoire de `size` octets consécutifs dans la zone de mémoire du tas.

- **Valeur retournée :**

  - Si l'allocation réussit, `malloc` retourne un pointeur sur le début du bloc alloué.
  - Si la place disponible est insuffisante ou si `size` vaut 0, `malloc` retourne `NULL`.

- **Remarque :**

  Les fonctions d'allocation dynamique retournent des pointeurs sur des `void`. Il faut donc opérer des conversions de types explicites pour utiliser ces zones mémoire en fonction du type des données qui y seront mémorisées.

Il peut être fait appel à une macro afin d'alléger l'écriture :

```
#define MALLOC(t, n) (t *) malloc(n * sizeof(t))

...

int *ptr;

ptr = MALLOC(int, 10);
/* equivalent:   ptr = (int *) malloc(10 * sizeof(int));
```

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>

main(void) {
  char *ptr;
  struct s_fiche { char nom[30];
                   int  numero;
                   struct s_fiche *suiv;
                 } *fiche;

                              /* demande d'allocation de 80 octets */
  ptr = malloc(80);
  if ( ptr == NULL) {
    fprintf(stderr,"Allocation memoire impossible\n");
    exit(1);
  }

  if (fiche = (struct s_fiche *) malloc(sizeof(struct s_fiche)) == NULL) {
    fprintf(stderr,"Allocation memoire impossible\n");
    exit(1);
  }

  free(fiche);              /* liberation de la memoire */
  free(ptr);
}
```

11.8.2 Libération mémoire (`free`)

- **Syntaxe :**

```
#include <stdlib.h>

void free(void *ptr);
```

- **Description :**

  La fonction `free` libère un bloc mémoire d'adresse de début `ptr`.

  Ce bloc mémoire a été précédemment alloué par une des fonctions `malloc`, `calloc`, ou `realloc`.

  Attention :

  - Il n'y a pas de vérification de la validité de `ptr`.
  - Ne pas utiliser le pointeur `ptr` après `free`, puisque la zone n'est plus réservée.
  - A tout appel de la fonction `malloc` (ou `calloc`) doit correspondre un et un seul appel à la fonction `free`.
  - Les appels pour libérer des blocs de mémoire peuvent être effectués dans n'importe quel ordre, sans tenir compte de celui dans lequel les blocs ont été obtenus à l'allocation.

- **Valeur retournée :** aucune

- **Exemple 1 :**

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *str = NULL, i;

    str = (int *) malloc(10 * sizeof(int));
    for (i=0; i<10; i++)
        str[i] = i * 10;
    printf("%d\n", i[9]);
    free(str);
    return 0;
 }
```

- **Exemple 2 :** illustration d'une erreur commise fréquemment

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *str;

    str = (char *) malloc(100 * sizeof(char));
    gets(str);  /* saisie d'une chaine de caracteres */

    /* suppression des espaces en tete de chaine */
    while ( *str == ' ')
      str++;
```

```
    /* free ne libere pas toute la zone allouee car ptr ne designe
       plus le debut de la zone memoire allouee par malloc */
    free(str);

    return 0;
}
```

11.8.3 Demande d'allocation mémoire (`calloc`)

- **Syntaxe :**

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

- **Description :**

  La fonction `calloc` réserve un bloc de taille `nelem x elsize` octets consécutifs. Le bloc alloué est initialisé à `0`.

- **Valeur retournée :**

  - Si succès, `calloc` retourne un pointeur sur le début du bloc alloué.
  - Si échec, `calloc` retourne `NULL` s'il n'y a plus assez de place ou si `nelem` ou `elsize` valent `0`.

- **Exemple :**

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *str = NULL;

    str = (char *) calloc(10, sizeof(int));
    printf("%d\n", str[9]);
    free(str);
    return 0;
 }
```

11.8.4 Demande de réallocation mémoire (`realloc`)

- **Syntaxe :**

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

- **Description :**

  La fonction `realloc` ajuste la taille d'un bloc à `size` octets consécutifs.

- **Arguments :**

  - `ptr` : pointeur sur le début d'un bloc mémoire créé par `malloc`, `calloc`, ou `realloc`. Si ce pointeur est `NULL`, `realloc` équivaut à `malloc`.
  - `size` : nouvelle taille du bloc en octets.

- **Valeur retournée :**

  - Si succès, cette fonction retourne l'adresse de début du bloc réalloué.

    Cette adresse peut avoir changé par rapport à celle fournie en argument. Dans ce cas, le contenu de l'ancien bloc est copié à la nouvelle adresse et l'ancienne zone est automatiquement libérée.
  - Si échec, (pas assez de place en mémoire ou `size` à `0`), `realloc` retourne la valeur `NULL`.

- **Exemple 1 :**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    char *str1=NULL, *str2=NULL;

    str1 = (char *) malloc(11);
    strcpy(str1, "ABCDEFGHIJ");

    str2 = (char *) realloc(str2, 20);

    printf("Adresse de str1 : %p\n", str1);
    printf("Adresse de str2 : %p\n", str2);

    str1 = (char *) realloc(str1, 100);

    printf("Nouvelle adresse de str1 : %p\n", str1);
    printf("Contenu de str1 : %s\n", str1);
    free(str1);
    free(str2);
    return 0;
}
/*-- resultat de l'execution ----------------------------
Adresse de str1 : 05E6
Adresse de str2 : 05F6
Nouvelle adresse de str1 : 060E
Contenu de str1 : ABCDEFGHIJ
------------------------------------------------------------*/
```

- **Exemple 2 :** saisie d'une chaîne de longueur variable

```
#include <stdio.h>
#include <stdlib.h>

/*************************************************************
 * Lecture d'une chaine de caracteres de longueur quelconque *
 *************************************************************/

char *lirechaine( void ) {
  int t = 16;                      /* taille du buffer                */
  char *buf = (char *) malloc(16); /* debut du buffer de lecture      */
  char *pos = buf;                 /* position d'ecriture ds le buffer */

  while ( (*pos = getchar()) != '\n' ) {
    pos++;
    if ( pos == buf + t ) {
                      /* on augmente la taille du buffer de 16 octets */
      buf = (char *) realloc(buf, t + 16);
      pos = buf + t;      /* le buffer peut etre a une autre adresse ! */
      t  += 16;
    }
  }
  *pos++ = '\0';
      /* on ajuste la taille du buffer a la taille reelle de la chaine */
  buf = (char *) realloc(buf, pos - buf);
  return buf;
}


main(void) {
  char *str1;

  printf("chaine: ");
  str1 = lirechaine();
  printf("chaine: %s\n");
  free(str1);
  return 0;
}
```

11.9 Opérations sur la mémoire

Ces fonctions sont similaires à certaines fonctions de la librairie des chaînes de caractères. Ces dernières opèrent sur des chaînes terminées par le caractère nul, tandis que les fonctions décritent ci après opèrent sur des zones de la mémoire de type indifférent.

11.9.1 Copie d'un bloc mémoire (`memcpy`)

- **Syntaxe :**

```
#include <string.h>

void *memcpy (void *dest, const void *srce, size_t n);
```

- **Description :** copie un bloc de `n` octet(s) de `srce` vers `dest`.

  Attention : si les zones `srce` et `dest` se chevauchent, le comportement de la fonction devient imprévisible...

- **Valeur retournée :** `memcpy` retourne la valeur de `dest`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int tab[2][5] = { { 1, 2, 3, 4, 5},
                      {11, 12, 13, 14, 15} };
    int temp[2][5];

    memcpy(temp, tab, sizeof(tab));
    printf("temp[1][4] = %d\n", temp[1][4]);
    return 0;
}
/* Resultat de l'execution : --------------------------------
temp[1][4] = 15
-------------------------------------------------------------*/
```

11.9.2 Copie limitée d'un bloc (`memccpy`)

- **Syntaxe :**

```
#include <string.h>

void *memccpy(void *dest, const void *srce, int c, size_t n);
```

- **Description :** copie un bloc de `n` octets de `srce` vers `dest`. La copie s'arrête dès que le caractère `c` est copié dans `dest` ou que `n` octets sont copiés dans `dest`.

- **Valeur retournée :** si `c` a été copié, `memccpy` renvoie un pointeur sur l'octet qui le suit dans `dest`, sinon, `memccpy` renvoie `NULL`.

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
```

```
    char *srce = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    char dest[80];
    char *ptr;

    ptr = (char *) memccpy(dest, srce, 'I', strlen(srce));

    if (ptr) {
       *ptr = '\0';  /* Attention, indispensable ... */
       printf("Trouve: %s\n", dest);
    }
    else
       printf("Pas trouve 'I' dans %s\n", srce);
    return 0;
}
/* Resultat de l'execution : ---------------------------------
Trouve: ABCDEFGHI
-------------------------------------------------------------*/
```

11.9.3 Copie d'un bloc (memmove)

- **Syntaxe :**

```
#include <string.h>

void *memmove(void *dest, const void *srce, size_t n);
```

- **Description :** copie un bloc de `n` octets de `srce` vers `dest`. Pour `memmove`, contrairement à `memcpy`, même si `srce` et `dest` se chevauchent, les octets de la zone en conflit sont copiés correctement.
- **Valeur retournée :** `memmove` retourne `dest`.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char *dest = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
   char *srce = "**************************";

   printf("dest avant 'memmove': %s\n", dest);
   memmove(dest, srce, 26);
   printf("dest apres 'memmove': %s\n", dest);
   return 0;
 }
/* Resultat de l'execution : ---------------------------------
dest apres 'memmove': **************************0123456789
-------------------------------------------------------------*/
```

11.9.4 Recherche d'un octet dans un bloc (memchr)

- **Syntaxe :**

```
#include <string.h>

void *memchr (const void *buffer, int c, size_t n);
```

- **Description :** `memchr` scrute les `n` premiers octets de `bufffer` à la recherche du caractère `c`.
- **Valeur renvoyée :** si succès, renvoie un pointeur sur la première occurrence de `c` dans `buffer`, sinon, renvoie `NULL`.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char ch[30];
    char *ptr;

    strcpy(ch, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    ptr = (char *) memchr(ch, 'I', strlen(ch));
    if (ptr)
       printf("Le caractere 'I' est en position: %d\n", ptr - ch);
    else
       printf("Le caractere 'I' n'est pas trouve\n");
    return 0;
}
/* Resultat de l'execution : ---------------------------------
Le caractere 'I' est en position: 8
------------------------------------------------------------*/
```

11.9.5 Comparaison de zones mémoire (memcmp)

- **Syntaxe :**

```
#include <string.h>

int memcmp (const void *buf1, const void *buf2, size_t n);
```

- **Description :** `memcmp` compare lexicographiquement les `n` premiers octets de `buf1` et `buf2`. La comparaison est entre types `unsigned char`.
- **Valeur renvoyée :** `memcmp` retourne une valeur:
  - négative si `buf1 < buf2`

- égale à 0 si `buf1 == buf2`
- positive si `buf1 > buf2`

- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char *buf1 = "ABCDE";
    char *buf2 = "abc";
    int etat = memcmp(buf1, buf2, strlen(buf2));

    if (etat > 0)
       printf("%s > %s\n", buf1, buf2);
    else
       if (etat == 0)
          printf("%s == %s\n", buf1, buf2);
       else
          printf("%s < %s\n", buf1, buf2);
    return 0;
}
/* Resultat de l'execution : --------------------------------
ABCDE < abc
------------------------------------------------------------*/
```

11.9.6 Initialisation d'une zone mémoire (`memset`)

- **Syntaxe :**

```
#include <string.h>

void *memset (void *buffer, int c, size_t n);
```

- **Description :** `memset` initialise les `n` premiers octets de `buffer` avec le caractère `c`.
- **Valeur retournée :** la valeur de `buffer`.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buf[] = "E.N.A.C.";

    printf("Buf avant 'memset': %s\n", buf);
    memset(buf, '*', strlen(buf));
    printf("Buf apres 'memset': %s\n", buf);
```

```
    return 0;
}
/* Resultat de l'execution : ---------------------------------
Buf avant 'memset': E.N.A.C.
Buf apres 'memset': ********
------------------------------------------------------------*/
```

11.9.7 Portabilité des applications BSD (`bcopy`, `bcmp`, `bzero`)

Pour des raisons de portabilité des applications écrites sous système BSD, un certain nombre de fonctions sont disponibles :

- **Syntaxe :**

```
#include <strings.h>    /* attention au s */

void bcopy(const char *buf1, char *buf2, int n);
int  bcmp(const char *buf1, const char *buf2, int n);
void bzero(char *buf, int n);
```

- **Description :**
  - `bcopy` copie `n` octets de la zone pointée par `buf1` dans la zone d'adresse `buf2`.
  - `bcmp` compare les `n` octets de la zone pointée par `buf1` avec ceux de la zone d'adresse `buf2`.
  - `bzero` met à zéro les `n` octets de la zone pointée par `buf`.

11.10 Les branchements hors fonction

- **Syntaxe :**

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int retval);
```

- **Description :**

  La macro `setjmp` capture l'état complet de la tâche courante dans `env` puis retourne la valeur 0.

  Un appel à la fonction `longjmp` avec `env` comme paramètre, restaure l'état de la tâche capturée par `setjmp` et cette dernière retourne la valeur `retval`.

- **Remarques :**
  - La macro `setjmp` doit **obligatoirement** être appelée avant la fonction `longjmp`.

- `longjmp` ne peut pas transmettre la valeur de retour `0`.
- La fonction ayant appelé `setjmp` et initialisé `env` doit rester active et ne peut se terminer avant l'appel à `longjmp`, faute de quoi le résultat est imprévisible...
- La macro `setjmp` permet de gérer les erreurs et exceptions et de revenir immédiatement d'une fonction profondément imbriquée.

- **Valeur retournée :**

  - `setjmp` renvoie la valeur `0` indiquant ainsi le retour du premier appel à la macro. Au retour d'un appel à `longjmp`, `setjmp` renvoie une valeur différente de zéro (le paramètre `retval` de la fonction `longjmp`).
  - `longjmp` ne renvoie rien.

- **Exemple :**

```
#include <stdio.h>
#include <setjmp.h>

void essai_jump(jmp_buf jb, int n);

main(void) {
    int v;
    jmp_buf jump;

    v = setjmp(jump);
    if (v != 0) {
       printf("Retour de longjmp avec la valeur %d\n", v);
       return v;
    }
    printf("Appel de la fonction essai_jump\n");
    essai_jump(jump, 1);
    return 0;
}

/* fonction recursive
   au 10eme appel recursif on sort immediatement */
void essai_jump(jmp_buf jb, int n) {
    if ( n == 10 )
      longjmp(jb, n);
    else
      essai_jump(jb, n+1);
}
/*-- resultat de l'execution ---------------------------
Appel de la fonction essai_jump
Retour de longjmp avec la valeur 10
------------------------------------------------------*/
```

11.11 Date et heure

11.11.1 Récupération de l'heure système (time)

- **Syntaxe :**

```
#include <time.h>

time_t time(time_t *t);
```

- **Description :** la fonction `time` nous donne le nombre de secondes écoulées depuis le 1er janvier 1970 à 0 heure GMT.

  Si le pointeur `t` est différent de `NULL`, `time` place cette valeur à cette adresse.

- **Valeur retournée :** le temps écoulé en secondes depuis le 1er janvier 1970 à 0 heure GMT.

  Une utilisation courante de cette valeur est d'initialiser le générateur de nombre aléatoire (c.f. `srand` page 142)

- **Exemple :**

```
#include <stdio.h>
#include <time.h>

main(void) {
  time_t t;

  srand( time(NULL) );
  time(&t);   /* ou      t = time(NULL);   */
  printf("Nbre de secondes depuis le 1er Janvier 1970 : %ld\n", t);

  return 0;
}
/*-- resultat de l'execution ---------------------------------
Nbre de secondes depuis le 1er Janvier 1970 : 787225446
------------------------------------------------------------*/
```

11.11.2 Conversion de la date et l'heure en une chaîne (ctime)

- **Syntaxe :**

```
#include <time.h>

char *ctime(const time_t *t);
```

Description :

La fonction `ctime` convertit la valeur `*t` en une chaîne de 26 caractères au format suivant :

`"NNN MMM jj hh:mm:ss AAAA\n\0"`

avec :

`NNN`	Nom du jour (Mon, Tue, Wed, etc...)
`MMM`	Nom du mois (Jan, Feb, Mar, etc...)
`jj`	Numéro du jour (1, 2, ..., 31)
`hh`	Heure (1, 2, ..., 24)
`mm`	Minutes (1, ..., 59)
`ss`	Secondes (1, ..., 59)
`AAAA`	Année

- **Valeur retournée :**

  Cette fonction renvoie un pointeur sur une chaîne de caractères contenant la date et l'heure.

- **Attention :**

  - C'est la même chaîne statique qui est retournée à chaque appel.
  - Le paramètre `t` est un pointeur [5].

- **Exemple :**

```
#include <stdio.h>
#include <time.h>

main(void) {
    time_t t;

    time(&t);
    printf("date et heure : %s*****\n", ctime(&t));
    return 0;
}
/*-- resultat de l'execution ---------------------------------
date et heure : Mon Dec 12 10:46:20 1994
*****
------------------------------------------------------------*/
```

11.11.3 Conversion d'une date et heure en une structure (`localtime`)

- **Syntaxe :**

[5] Cela vient du temps où une fonction ne pouvait pas recevoir en paramètre une valeur de type `long`

```
#include <time.h>

struct tm *localtime(const time_t *t);
```

- **Description :**

  Cette fonction convertit la valeur reçue en une structure de type `struct tm` contenant les composants temporels et en appliquant des corrections de fuseau horaire et d'heure d'été ("daylight saving time").

- **La structure `struct tm` :**

  Structure définissant des subdivisions du temps :

```
struct tm {
  int tm_sec;   /* Secondes */
  int tm_min;   /* Minutes */
  int tm_hour;  /* Heures (0 - 23) */
  int tm_mday;  /* Quantieme du mois (1 - 31) */
  int tm_mon;   /* Mois (0 - 11) */
  int tm_year;  /* An (annee calendaire - 1900) */
  int tm_wday;  /* Jour de semaine (0 - 6  Dimanche = 0) */
  int tm_yday;  /* Jour dans l'annee (0 - 365) */
  int tm_isdst; /* 1 si "daylight saving time" */
};
```

- **Valeur retournée :**

  `localtime` retourne un pointeur sur une structure de type `struct tm` contenant les différents composants temporels.

- **Attention :**

  Cette structure est de classe statique et elle est écrasée à chaque appel.

- **Exemple :**

```
#include <stdio.h>
#include <time.h>

main(void) {
    time_t t;
    struct tm *tb;

    t = time(NULL);
    tb = localtime(&t);
    printf("date et heure : %s\n", ctime(&t));
    printf("Le numero du mois est: %d\n", tb->tm_mon);

    return 0;
}
```

```
/*-- resultat de l'execution ----------------------------
date et heure : Mon Dec 12 10:47:06 1994

Le numero du mois est: 11
-------------------------------------------------------*/
```

11.11.4 Conversion d'une date et heure au standard GMT (gmtime)

- **Syntaxe :**

```
#include <time.h>

struct tm *gmtime(const time_t *t);
```

- **Description :**

`gmtime` convertit directement vers l'heure GMT (Greenwich Mean Time) contrairement à `localtime` qui applique des corrections de fuseau horaire et d'heure d'été.

- **Valeur retournée :**

`localtime` retourne un pointeur sur une structure de type `struct tm` contenant les différents composants temporels.

- **Attention :**

Cette structure est de classe statique et elle est écrasée à chaque appel.

11.11.5 Conversion de la date et l'heure en une chaîne (asctime)

- **Syntaxe :**

```
#include <time.h>

char *asctime(const struct tm *tblock);
```

- **Description :**

`asctime` convertit le contenu de la structure `*tblock` en une chaîne de 26 caractères au format suivant :

`"NNN MMM jj hh:mm:ss AAAA\n\0"`

avec :

`NNN`	Nom du jour (Mon, Tue, Wed, etc...)
`MMM`	Nom du mois (Jan, Feb, Mar, etc...)
`jj`	Numéro du jour (1, 2, ..., 31)
`hh`	Heure (1, 2, ..., 24)
`mm`	Minutes (1, ..., 59)
`ss`	Secondes (1, ..., 59)
`AAAA`	Année

- Valeur retournée :

  Cette fonction renvoie un pointeur sur une chaîne de caractères contenant la date et l'heure.

- **Attention :**

  C'est la même chaîne statique qui est retournée à chaque appel.

- **Exemple :**

```
#include <stdio.h>
#include <time.h>

main(void) {
    time_t t;
    struct tm *tb;

    t = time(NULL);
    tb = localtime(&t);
    printf("date et heure : %s", ctime(&t));
    printf("date et heure : %s", asctime(tb));

    return 0;
}
/*-- resultat de l'execution ----------------------------
date et heure : Mon Dec 12 10:47:52 1994
date et heure : Mon Dec 12 10:47:52 1994
-------------------------------------------------------*/
```

11.11.6 Calcule le délai séparant deux instants (`difftime`)

- **Syntaxe :**

```
#include <time.h>

double difftime(time_t time2, time_t time1);
```

- **Description :**

  La fonction `difftime` calcule le temps écoulé en secondes entre `time2` et `time1`.

- **Valeur retournée :**

  Un nombre de type double représentant la différence en secondes.

- **Exemple :**

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>   /* sleep() */
```

```
main(void) {
    time_t t1, t2;

    t1 = time(NULL);
    sleep(2);       /* attente de 2 secondes */
    t2 = time(NULL);

    printf("difference : %lf secondes\n", difftime(t2, t1) );
    return 0;
}
/*-- resultat de l'execution -------------------------
difference : 2.000000 secondes
-----------------------------------------------------*/
```

11.11.7 Conversion du temps au format du calendrier (`mktime`)

- **Syntaxe :**

```
#include <time.h>

time_t mktime(struct tm *t);
```

- **Description :**

  La fonction `mktime` convertit la date et l'heure contenues dans la structure `*t` en valeurs au format retourné par la fonction `time`.

- **Valeur retournée :** une valeur de type `time_t` si le temps `t` a pu être converti sinon `mktime` retourne la valeur `-1`.

- **Exemple :**

```
#include <stdio.h>
#include <time.h>

main(void) {
    struct tm t;
    time_t tps;
    int jour, mois, an;

    printf("Jour : "); scanf("%d", &jour);
    printf("Mois : "); scanf("%d", &mois);
    printf("Annee: "); scanf("%d", &an);

    t.tm_year = an - 1900;
    t.tm_mon  = mois - 1;
    t.tm_mday = jour;
    t.tm_hour = t.tm_min  = t.tm_sec  = 0;
```

```
        tps = mktime(&t);

        printf("Date et heure: %s\n", ctime(&tps));
        return 0;
    }
    /*-- resultat de l'execution ------------------------
    Jour : 12
    Mois : 12
    Annee: 1994
    Date et heure: Mon Dec 12 00:00:00 1994
    -------------------------------------------------*/
```

11.12 Fiabilisation d'un programme

La librairie standard ne gère pas les erreurs pouvant survenir en cours d'exécution.

Certaines fonctions renvoient un code d'erreur, qu'en général les (mauvais) programmeurs ignorent, fabriquant ainsi des programmes incapables de faire face au moindre problème.

Pour fiabiliser un programme, un contrôle systématique des valeurs de retour est indispensable. Ce travail est tellement fastidieux qu'il n'est jamais complètement fait et il rend les programmes longs et peu lisibles.

Exemple :

```
if ( (f_in = fopen("essai", "r")) == NULL ) {
  fprintf(stderr,"Erreur a l'ouverture\n");
  exit(1);
}
if ( (f_out = fopen("tempo.tmp", "w")) == NULL ) {
  fprintf(stderr,"Erreur a la creation\n");
  exit(1);
}
if ( (buffer = malloc(TAILLE)) == NULL ) {
  fprintf(stderr,"Erreur a l'allocation memoire\n");
  exit(1);
}
if ( fread(buffer, 1, TAILLE, f_in) == 0 ) {
  fprintf(stderr,"Erreur a la lecture\n");
  exit(1);
}
if ( fwrite(buffer, 1, TAILLE, f_out) != TAILLE ) {
  fprintf(stderr,"Erreur a l'ecriture\n");
  exit(1);
}
if ( fclose(f_in) == EOF ) {
  fprintf(stderr,"Erreur a la fermeture\n");
  exit(1);
}
```

```
if ( fclose(f_out) == EOF ) {
  fprintf(stderr,"Erreur a la fermeture\n");
  exit(1);
}
```

11.12.1 Surdéfinition des fonctions

Il est plus rentable de redéfinir les fonctions à risques, ou d'utiliser des macros :

Exemple mêlant des surdéfinitions et des macros :

```
#define erreur(msg) {      \
     fprintf(stderr, "%s\n", msg);   \
     exit(1);                        \
   }

#define MALLOC(ptr, nbre, type)                                          \
   if ( (ptr = (type *) malloc( nbre * sizeof(type) )) == NULL )    \
      erreur("allocation memoire impossible");

FILE *Fopen(char *nom, char *mode) {
  FILE *f;

  if ( (f = fopen(nom, mode)) == NULL )
    erreur("ouverture impossible");
  return f;
}

int Fread(void *buffer, int size, int nbre, File *f) {
  int n;
  if ( (n = fread(buffer, size, nbre, f)) == 0 )
    erreur("erreur a la lecture");
  return n;
}

void Fwrite(void *buffer, int size, int nbre, File *f) {
  if ( fwrite(buffer, size, nbre, f)) != nbre )
    erreur("erreur a l'ecriture");
}

void Fclose(File *f) {
  if ( fclose(f) == EOF )
    erreur("erreur a la fermeture");
}
```

le programme devient :

```
Fopen("essai", "r");
Fopen("tempo.tmp", "w");
MALLOC(buffer, TAILLE, char);
```

```
Fread(buffer, 1, TAILLE, f_in);
Fwrite(buffer, 1, TAILLE, f_out);
Fclose(f_in);
Fclose(f_out);
```

Ces surdéfinitions et macros peuvent être mises dans un module et compilées séparément.

11.12.2 Macro de diagnostic

Pendant l'étape de mise au point d'un programme, il peut être utile d'insérer des points de contrôle, au prix de quelques lignes de code supplémentaires.

En particulier, l'une des premières choses à vérifier est la validité des indices d'un tableau. Ce type d'erreur n'est détectable ni à la compilation ni à l'exécution. Le compilateur C ne générant pas de code de contrôle (performance oblige).

La macro **assert** peut aider à résoudre ce problème.

La macro `assert`

- **Syntaxe :**

```
#include <assert.h>
void assert(int expression);
```

- **Description :** il s'agit d'une macro qui est évaluée comme une instruction conditionnelle.

  Si `expression` retourne zéro, `assert` affiche un message d'erreur et appelle la fonction `abort` pour finir le programme.

  Le message d'erreur affiché sur `stderr` est de la forme :

  ```
  Assertion failed: expression, file __FILE__, line __LINE__
  ```

  avec :

  `__FILE__` est une macro donnant le nom du fichier source

  `__LINE__` est une macro donnant le numéro de ligne courante de la macro `assert`

  Si la directive `#define NDEBUG` est placée dans le texte source avant `#include <assert.h>`, les instructions `assert` sont désactivées et ne ralentissent ni n'augmentent le code généré.

  On peut aussi utiliser le fait que `#define NDEBUG` soit présent ou pas, pour afficher des valeurs particulières du programme en utilisant la directive du préprocesseur `#ifndef NDEBUG`.

- **Valeur retournée :** aucune

- **Exemple :**

```
#include <assert.h>
#include <stdio.h>

#define TAILLE 512

int calcul(int tab[], int i) {
    assert(i >= 0 && i < TAILLE);
    return 2 * tab[i] + 5;
}

int main(void) {
  int tableau[TAILLE];

#ifndef NDEBUG
  printf("DEBUG: tableau[0] = %d\n", tableau[0]);
#endif

  printf("%d\n", calcul(tableau, 1024));

  return 0;
}
/*-- resultat de l'execution ----------------------------
DEBUG: tableau[0] = 3456
Assertion failed: i >= 0 && i < TAILLE, file assert.c, line 7
IOT trap (core dumped)
------------------------------------------------------------*/
```

Annexe A

L'utilitaire make

A.1 Généralités

Make est un utilitaire qui permet de gérer des programmes de grande ampleur comportant de nombreux fichiers sources et fichiers objets. Il permet de générer une application d'après une description qui en est faite dans un fichier spécial, appelé *fichier make* ou *makefile*.

Ce fichier décrit les différentes dépendances des fichiers sources et les commandes qui sont à exécuter pour les générer si besoin. Cette génération est faite de manière optimale, car seules les opérations nécessaires sont effectuées. Ainsi, il n'y a pas de recompilation d'un fichier source si celui-ci et ses fichiers inclus n'ont pas été modifiés.

make se base pour effectuer cette génération sur la date et l'heure de création des fichiers objets par rapport à la date et l'heure de modification des fichiers sources et de ses dépendances : si l'un de ces derniers est plus récent que le fichier objet, *make* sait qu'il y a eu des modifications et que le fichier objet (appelé **cible**) doit être recompilé.

A.2 Principe de fonctionnement

Si *make* est lancé sans argument, il recherche la première cible du *makefile*. Chacune des dépendances de cette première cible devient une cible pour *make* et ainsi de suite récursivement.

Cette scrutation lui permet de connaître l'ensemble de l'arbre des dépendances.

make exécute, si nécessaire, les commandes lui permettant de reconstruire les cibles à partir du bas de l'arbre et en remontant jusqu'à sa racine.

Si *make* est lancé avec un argument ayant pour valeur le nom d'une cible, il commence son exécution à partir de cette cible racine.

A.3 Fichier *makefile*

Le fichier contenant la description des dépendances est par défaut nommé `Makefile` ou `makefile`. Il est organisé de la façon suivante :

```
cible ... : dependance1 ...
```

```
commandes
...
```

cible désigne le nom du fichier cible à mettre à jour ou à créer. Le nom d'une cible est toujours suivi du caractère ':' (deux-points).

Une cible peut ne pas avoir de dépendance, dans ce cas, elle représente un point d'entrée pour exécuter une suite de commandes lorsque l'on la nomme explicitement.

commandes désigne les commandes qui seront exécutées, par un shell, pour mettre à jour la cible.

Une ligne de commandes commence toujours par un espace ou une tabulation.

Exemple :

```
gestfich : main.o edit.o list.o
        cc -Aa -g -o gestfich main.o edit.o list.o -L/usr/local/lib
main.o : defs.h main.c
        cc -Aa -g -c main.c
edit.o : defs.h edit.c
        cc -Aa -g -c edit.c
list.o : defs.h list.c
        cc -Aa -g -c list.c

# nettoyage des fichiers inutiles
clean :
        rm -f gestfich main.o edit.o list.o core *~ a.out
```

Tous les caractères suivant un `#` sont considérés comme un commentaire.

Pour utiliser ce `makefile` pour générer le fichier `gestfich`, il faut taper `make` sur la ligne de commande du shell.

Le fichier `gestfich` dépend des modules objets `main.o`, `edit.o` et `list.o`.

`main.o` dépend des fichiers `defs.h` et `main.c`.

Si un de ces derniers est modifié, la commande `cc -c main.c` est exécutée pour régénérer le fichier `main.o` et ainsi de suite pour `edit.o` et `list.o`.

Si l'un des fichiers dont dépend `gestfich` a été régénéré, la commande `cc -o gestfich main.o edit.o list.o` est exécutée.

Si sur la ligne de commande du shell, la commande `make clean` est tapée, la commande `rm` est exécutée.

A.4 Macros

Une macro est un nom qui représente une chaîne de caractères. Cela permet de simplifier l'écriture des *makefile*.

A.4.1 Initialisation d'une macro

Syntaxe :

```
NOM_MACRO = chaine_de_caracteres
```

Généralement les macros sont nommés en majuscules.

Exemples :

```
OBJ = main.o edit.o list.o
CC  = cc -Aa -g
LIBDIR = -L/usr/local/lib
```

A.4.2 Appel d'une macro

Syntaxe :

```
$(NOM_MACRO)
```

Ainsi les 2 premières lignes du fichier `makefile` exemple de la page 170 peuvent s'écrire sous la forme suivante :

```
gestfich : $(OBJ)
        $(CC) -o gestfich $(OBJ) $(LIBDIR)
```

Remarques :

- si la macro n'est pas définie, *make* remplace son invocation par la chaîne vide.
- il peut y avoir une macro dans une macro.
- les variables de l'environnement sont connues et utilisables dans le fichier `makefile`.

```
install :
       cp gestfich $(HOME)/bin
```

A.4.3 Macros prédéfinies

- `$$` correspond au caractère `$`
- `$@` correspond au nom complet de la cible courante
- `$*` correspond au nom de base (sans extension) de la cible courante
- `$<` correspond au nom complet de la première cible courante

A.5 Règles implicites

Des commandes implicites seront appelées automatiquement si aucune commande n'est spécifiée. Ainsi, *make* sait qu'on obtient un fichier "`.o`" en compilant un fichier "`.c`" par la commande `cc -c`.

Ainsi le début du fichier `makefile` exemple de la page 170 peut s'écrire sous la forme suivante :

```
gestfich : $(OBJ)
        $(CC) -o gestfich $(OBJ) $(LIBDIR)
main.o : defs.h main.c
edit.o : defs.h edit.c
list.o : defs.h list.c
```

La syntaxe générale d'une règle implicite est :

```
.source_extension.target_extension:
        commande
        ...
```

La règle implicite :

```
.c.o:
     $(CC) $(CFLAGS) -c $<
```

permet la génération de fichiers "`.o`" à partir de fichiers "`.c`".

A.6 Principales options de la commande *make*

- `-f filename` : permet d'indiquer le nom du fichier *makefile*. Si cette option est omise, *make* recherche successivement un fichier dont le nom est *makefile* puis *Makefile*.
- `-n` : affiche uniquement les commandes sans les exécuter. Ceci est utile pour mettre au point les fichiers *makefile*.
- `-s` : mode silencieux. *make* n'affiche pas les commandes avant de les exécuter.
- `-p` : liste les macros prédéfinies ainsi que les règles implicites.
- `macro_name=valeur` : définition d'une macro.

```
# Make exemple generique sous HPUX

# definition des macros
CC = cc                         # avec optimisation du code
DEFS = -Aa -D_HPUX_SOURCE       # compilation C-ANSI
CFLAGS = $(DEFS) -g -O

INC = -I/usr/local/include/Motif1.1 -I/usr/local/include/X11R4 \
      -I/usr/local/include -I/usr/local/X11/usr/include

LIBS = -L/usr/lib/X11R4 -L/usr/local/lib  -L/usr/local/lib/Motif1.1 \
       -L/usr/local/lib/X11R4 -L/usr/local/X11/usr/lib \
       -lmalloc -lXm -lXt -lX11 -lm -lXaw -lXmu -lXext

MAIN    =  xkeycaps             # Nom du programme executable

# Nom des sources
SRCS = xkeycaps.c KbdWidget.c KeyWidget.c info.c actions.c \
       commands.c guess.c

# Nom des objets
OBJS = xkeycaps.o KbdWidget.o KeyWidget.o info.o actions.o \
       commands.o guess.o

all: $(MAIN)

# regles implicites :
.c.o:
        $(CC) $(INC) $(DEFS) -c $<

# regles explicites :
$(MAIN): $(OBJS)
        $(CC) -o $(MAIN) $(LDFLAGS) $(OBJS) $(LIBS)

clean:
        rm -f $(OBJS) core a.out *~
```

Figure A.1: Exemple de fichier *makefile*

Annexe B

Le debugger xdb

B.1 Description

xdb est un debugger pour les langages C, Fortran, Pascal, et C++. Ce document ne présente que quelques aspects de xdb pour pouvoir l'utiliser rapidement et facilement. Les commandes essentielles sont fournies par des exemples (pour le langage C). L'information complète peut être obtenue par

```
man xdb
```

Pour pouvoir debugger en travaillant au niveau du code source (plus agréable qu'au niveau instructions machine ...), il faut avoir mis l'option `-g` dans la commande de compilation; par exemple :

```
cc -g mon_prog.c -o mon_prog
```

B.2 Lancement de xdb

Pour invoquer xdb sur le programme `mon_prog` faire :

```
xdb  mon_prog
```

Le debugger est ainsi lancé et le programme `mon_prog` est chargé. Les informations sont présentées dans une fenêtre séparée en deux parties:

- Celle du haut visualise le code source de `mon_prog`. L'index ">" indique la première instruction exécutable.
- Celle du bas est la zone où les commandes du debugger seront rentrées.

B.3 Visualisation du source

Plusieurs commandes sont intéressantes pour modifier la taille ou le contenu de la partie de fenêtre visualisant le code source.

`w 20`	fixe la taille de la zone de source qui vaut maintenant 20 lignes.
`v`	scroll du code source de la taille de la zone.
`v 110`	visualise la ligne 110 du source au milieu de la fenêtre.
`+ 15`	scroll avant de 15 lignes.
`- 10`	scroll arrière de 10 lignes.
`/ma_chaine`	cherche la prochaine chaine `ma_chaine` et visualise l'endroit trouvé.
`?ma_chaine`	même recherche mais en arrière.
`n`	répète la dernière recherche.
`N`	répète en arrière la dernière recherche.

B.4 Breakpoints

`b`	positionne un breakpoint à la ligne courante.
`b 75`	positionne un breakpoint à la ligne 75.
`b 75\10`	positionne un breakpoint à la ligne 75, mais qui ne sera reconnu qu'au dixième passage.

Quand un breakpoint sera rencontré à l'exécution, le déroulement du programme sera interrompu et le debugger affichera l'adresse d'arrêt puis se mettra en attente de commandes : généralement visualisation du contenu des variables ou modification de leur valeur. On peut cependant faire exécuter automatiquement des actions quand un breakpoint est rencontré :

```
b 80  {Q;p ma_variable;if (*mon_ptr ==-1) {"Aie aie aie"} {c} }
```

Cette commande positionne un breakpoint ligne 80 auquel est associée une liste (comprise entre { }) de commandes séparées par des ";"

`Q`	"Quiet" pour ne pas afficher l'adresse d'arrêt.
`p ma_variable`	affichage de la valeur de `ma_variable`.
`if (*mon_ptr ==-1)`	si la valeur pointée par `mon_ptr` vaut -1 alors :
`{"Aie aie aie"}`	écrire ce message
`{c}`	sinon continuer (reprendre l'exécution).

`abc {p strlen(ma_chaine)}`	la commande fournie en argument est exécutée à chaque breakpoint. (ici écriture de la longueur de la chaine `ma_chaine`).
`dbc`	annulation de la commande associée à chaque breakpoint.
`lb`	liste des breakpoints.
`db 2`	efface le breakpoint numéro 2. (Les breakpoints sont numérotés dans l'ordre où ils sont crées. On peut voir leur numéro grâce à la commande `lb`).
`bp`	met un breakpoint à l'entrée de chaque procédure.
`bpx`	met un breakpoint à la sortie de chaque procédure.
`bpt`	met un breakpoint à l'entrée et à la sortie de chaque procédure.
`dbp`	efface les breakpoints mis par l'une des 3 commandes ci−dessus.
`bc 3 10`	demande que le breakpoint 3 ne soit activé qu'au dixième passage.

B.5 Contrôles d'exécution

`r`	lance l'exécution du programme donné en argument à l'invocation de xdb.
`r arg1 arg2`	lance le même programme en lui fournissant les arguments `arg1` et `arg2`. Si ce n'est pas la première commande d'exécution, le process en cours est d'abord tué et un autre relancé avec par défaut les mêmes arguments que lors de la dernière commande.
`c`	continue l'exécution après un arrêt sur un breakpoint.
`c 75`	continue l'exécution et positionne un breakpoint à la ligne 75.
`s`	exécute une instruction (une ligne). Quand la dernière commande est `s`, à chaque *Return*, un pas est exécuté.
`s 3`	exécute 3 lignes.
`S`	idem `s` mais considère un appel de procédure comme une instruction sans faire du pas à pas dedans.
`g 50`	modifie le déroulement normal du programme : se branche en ligne 50.
`g +2`	se branche 2 lignes plus loin.
`k`	tue le process en cours.

B.6 Visualisation des variables

`p ma_var`	visualise le contenu de `ma_var` avec un format par défaut qui dépend du type de `ma_var`.
`p tab[5]`	visualise le contenu de `tab[5]`. Si maintenant on fait simplement *Return*, on visualisera `tab[6]`.
`p *struct_ptr`	visualise le contenu des éléments de la structure pointée par `struct_ptr`.
`p struct_ptr->data`	visualise le contenu du champs `data` de la structure pointée par `struct_ptr`.
`p *(&.+10)`	visualise ce qu'il y a 10 octets plus loin que la dernière valeur visualisée.
`p :ma_var`	visualise le contenu de la variable **globale** `ma_var` (si ambiguïté)
`p ma_proc:ma_var`	visualise le contenu de la variable `ma_var` de la procédure `ma_proc`.
`p ma_proc:3:ma_var`	visualise le contenu de la variable `ma_var` de la procédure `ma_proc` au troisième niveau d'appel (cas de la récursion).
`p ma_var\x`	visualise `ma_var` en **hexadécimal**.
`p ma_var\o`	visualise `ma_var` en **octal**.
`p ma_var\b`	visualise `ma_var` en **binaire**.
`p ma_var\d`	visualise `ma_var` en **décimal**.
`p ma_var?x`	visualise **l'adresse** de `ma_var`.
`p mon_adr\p`	visualise le nom de la procédure où se trouve l'adresse `mon_adr`.
`l`	liste les variables locales de la procédure courante.
`l ma_proc`	liste les variables de la procédure `ma_proc`.
`l ma_proc:2`	liste les variables de la procédure `ma_proc` au niveau d'appel 2 (récursion).
`lg`	liste les variables globales.
`lp`	liste les procédures.

`lp deb`	liste les procédures dont le nom commence par `deb`. Même principe pour `lg`.

B.7 Modification d'une variable

`p ma_var = nouv_valeur`	affecte `nouv_valeur` à `ma_var`.
`p j += 17`	`j` est incrémentée de 17.

B.8 Commandes diverses

`q`	quitte le debugger après demande de confirmation.
`t`	affiche la pile.
`L`	affiche la prochaine ligne à exécuter.
`V`	met à jour la fenêtre source pour voir la dernière ligne exécutée au centre.
`ss sauv_fic`	sauvegarde les breakpoints, macros et assertions pour une invocation ultérieure de xdb (par la commande `xdb mon_prog -R sauv_fic`). Mais attention si entre 2 appels, modification du source ....
`def ma_macro p alpha;p beta\x;p gamma[10]/2`	création d'une macro correspondant à 3 commandes d'affichage. Pour l'utiliser, faire une fois : `tm` pour activer les macros, puis par exemple positionner un breakpoint avec : `b 120 ma_macro`
`a {if (y != 10) {x} }`	création d'une assertion.

Une assertion est une liste de commandes à exécuter à **chaque** instruction ($\Rightarrow$ attention au temps de calcul ...). Dans l'exemple ci–dessus, on teste avant chaque instruction la valeur de `y` et on stoppe quand `y` ne vaut plus 10. Cette fonctionnalité s'utilise essentiellement pour débusquer une instruction sournoise qui vient sauvagement corrompre une bonne et honnête variable, souvent par débordement inapproprié d'une indexation ...

B.9 Debugger après un `core dump`

Si l'exécution d'un programme en dehors du debugger génère un "core dump", on peut essayer d'en comprendre la raison en examinant le fichier `core`. Pour ce faire, on sauvegarde le fichier `core` (pour ne pas l'écraser si nouveau "core dump"), puis on lance xdb :

```
mv core mon_core
xdb mon_prog mon_core
```

On peut maintenant grâce aux commandes de xdb, scruter l'état du programme avant le problème (bien sûr, pas de commande `r, c, s` ou `b` car l'exécution n'est plus possible). Mais on peut notamment exploiter la pile (commande `t`) ainsi que la valeur des variables au moment du crash.

Index

Symboles

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X