Le langage C++

Henri Garreta

Table des matières

31 mai 2006

`henri.garreta@luminy.univ-mrs.fr`

1 Eléments préalables

Ce document est le support du cours sur le langage C++, considéré comme une extension de C (tel que normalisé par l'ISO en 1990), langage présumé bien connu.

Attention, la présentation faite ici est déséquilibrée : des concepts importants ne sont pas expliqués, pour la raison qu'ils sont réalisés en C++ comme en C, donc supposés acquis. En revanche, nous insistons sur les différences entre C et C++ et notamment sur tous les éléments « orientés objets » que C++ ajoute à C.

Cette première section expose un certain nombre de notions qui, sans être directement liés à la méthodologie objets, font déjà apparaître C++ comme une amélioration notable de C.

1.1 Placement des déclarations de variables

En C les déclarations de variables doivent apparaître au début d'un bloc. En C++, au contraire, on peut mettre une déclaration de variable partout où on peut mettre une instruction[1].

Cette différence paraît mineure, mais elle est importante par son esprit. Elle permet de repousser la déclaration d'une variable jusqu'à l'endroit du programme où l'on dispose d'assez éléments pour l'initialiser. On lutte aussi contre les variables « déjà déclarées mais non encore initialisées » qui sont un important vivier de bugs dans les programmes.

Exemple, l'emploi d'un fichier. Version C :

```
{
    FILE *fic;
    ...
    obtention de nomFic, le nom du fichier à ouvrir
    Danger ! Tout emploi de fic ici est erroné
    ...
    fic = fopen(nom, "w");
    ...
    ici, l'emploi de fic est légitime
    ...
}
```

Version C++ :

```
{
    ...
    obtention de nomFic, le nom du fichier à ouvrir
    ici pas de danger d'utiliser fic à tort
    ...
    FILE *fic = fopen(nom, "w");
    ...
    ici, l'emploi de fic est légitime
    ...
}
```

1.2 Booléens

En plus des types définis par l'utilisateur (ou classes, une des notions fondamentales de la programmation orientée objets) C++ possède quelques types qui manquaient à C, notamment le type booléen et les types références.

Le type `bool` (pour booléen) comporte deux valeurs : `false` et `true`. Contrairement à C :

- le résultat d'une opération logique (`&&`, `||`, etc.) est de type booléen,
- là où une condition est attendue on doit mettre une expression de type booléen et
- il est déconseillé de prendre les entiers pour des booléens et réciproquement.

Conséquence pratique : n'écrivez pas « `if (x) ...` » au lieu de « `if (x != 0) ...` »

1.3 Références

1.3.1 Notion

A côté des pointeurs, les références sont une autre manière de manipuler les adresses des objets placés dans la mémoire. Une référence est un pointeur géré de manière interne par la machine. Si T est un type donné, le

[1] Bien entendu, une règle absolue reste en vigueur : une variable ne peut être utilisée qu'après qu'elle ait été déclarée.

type « référence sur T » se note T`&`. Exemple :

```
int i;
int & r = i;            // r est une référence sur i
```

Une valeur de type référence est une adresse mais, hormis lors de son initialisation, toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse. Il en découle qu'il est obligatoire d'initialiser une référence lors de sa création ; après, c'est trop tard :

```
r = j;                  // ceci ne transforme pas r en une référence
                        // sur j mais copie la valeur de j dans i
```

1.3.2 Références paramètres des fonctions

L'utilité principale des références est de permettre de donner aux fonctions des paramètres modifiables, sans utiliser explicitement les pointeurs. Exemple :

```
void permuter(int & a, int & b) {
    int w = a;
    a = b; b = w;
    }
```

Lors d'un appel de cette fonction, comme

```
permuter(t[i], u);
```

ses paramètres formels `a` et `b` sont initialisés avec les adresses des paramètres effectifs `t[i]` et `u`, mais cette utilisation des adresses reste cachée, le programmeur n'a pas à s'en occuper. Autrement dit, à l'occasion d'un tel appel, `a` et `b` ne sont pas des variables locales de la fonction recevant des copies des valeurs des paramètres, mais d'authentiques synonymes des variables `t[i]` et `u`. Il en résulte que l'appel ci-dessus permute effectivement les valeurs des variables `t[i]` et `u`[2].

1.3.3 Fonctions renvoyant une référence

Il est possible d'écrire des fonctions qui renvoient une référence comme résultat. Cela leur permet d'être le membre gauche d'une affectation, ce qui donne lieu à des expressions élégantes et efficaces.

Par exemple, voici comment une fonction permet de simuler un tableau indexé par des chaînes de caractères[3] :

```
char *noms[N];
int ages[N];

int & age(char *nom) {
    for (int i = 0; i < N; i++)
        if (strcmp(nom, noms[i]) == 0)
            return ages[i];
}
```

Une telle fonction permet l'écriture d'expressions qui ressemblent à des accès à un tableau dont les indices seraient des chaînes :

```
age("Amélie") = 20;
```

ou encore

```
age("Benjamin")++;
```

1.3.4 Références sur des données constantes

Lors de l'écriture d'une fonction il est parfois souhaitable d'associer l'efficacité des arguments références (puisque dans le cas d'une référence il n'y a pas recopie de la valeur de l'argument) et la sécurité des arguments par valeur (qui ne peuvent en aucun cas être modifiés par la fonction).

Cela peut se faire en déclarant des arguments comme des références sur des objets immodifiables, ou constants, à l'aide du qualifieur `const` :

[2] Notez qu'une telle chose est impossible en C, où une fonction appelée par l'expression `permuter(t[i], u)` ne peut recevoir que des *copies* des valeurs de de `t[i]` et `u`.

[3] C'est un exemple simpliste, pour faire court nous n'y avons pas traité le cas de l'absence de la chaîne cherchée.

```
void uneFonction(const unType & arg) {
    ...
    ici arg n'est pas une recopie de l'argument effectif (puisque référence)
    mais il ne peut pas être modifié par la fonction (puisque const)
    ...
}
```

1.3.5 Références ou pointeurs ?

Il existe quelques situations, nous les verrons plus loin, où les références se révèlent indispensables. Cependant, la plupart du temps elles font double emploi avec les pointeurs et il n'existe pas de critères simples pour choisir des références plutôt que des pointeurs ou le contraire.

Par exemple, la fonction `permuter` montrée à la page précédente peut s'écrire aussi :

```
void permuter(int *a, int *b) {
    int w = *a;
    *a = *b; *b = w;
    }
```

son appel s'écrit alors

```
permuter(&t[i], &u);
```

Attention On notera que l'opérateur `&` (à un argument) a une signification très différente selon le contexte dans lequel il apparaît :

- employé dans une déclaration, comme dans
  ```
  int &r = x;
  ```
  il sert à indiquer un type référence : « `r` est une référence sur un `int` »
- employé ailleurs que dans une déclaration il indique l'opération « obtention de l'adresse », comme dans l'expression suivante qui signifie « affecter l'adresse de `x` à `p` » :
  ```
  p = &x;
  ```
- enfin, on doit se souvenir qu'il y a en C++, comme en C, un opérateur `&` binaire (à deux arguments) qui exprime la conjonction bit-à-bit de deux mots-machine.

1.4 Fonctions en ligne

Normalement, un appel de fonction est une *rupture de séquence* : à l'endroit où un appel figure, la machine cesse d'exécuter séquentiellement les instructions en cours ; les arguments de l'appel sont disposés sur la pile d'exécution, et l'exécution continue ailleurs, là où se trouve le code de la fonction.

Une fonction en ligne est le contraire de cela : là où l'appel d'une telle fonction apparaît il n'y a pas de rupture de séquence. Au lieu de cela, le compilateur remplace l'appel de la fonction par le corps de celle-ci, en mettant les arguments affectifs à la place des arguments formels.

Cela se fait dans un but d'efficacité : puisqu'il n'y a pas d'appel, pas de préparation des arguments, pas de rupture de séquence, pas de retour, etc. Mais il est clair qu'un tel traitement ne peut convenir qu'à des fonctions fréquemment appelées (si une fonction ne sert pas souvent, à quoi bon la rendre plus rapide ?) de petite taille (sinon le code compilé risque de devenir démesurément volumineux) et rapides (si une fonction effectue une opération lente, le gain de temps obtenu en supprimant l'appel est négligeable).

En C++ on indique qu'une fonction doit être traitée en ligne en faisant précéder sa déclaration par le mot `inline` :

```
inline int abs(int x) {
    return x >= 0 ? x : -x;
    }
```

Les fonctions en ligne de C++ rendent le même service que les macros (avec arguments) de C, mais on notera que les fonctions en ligne sont plus fiables car, contrairement à ce qui se passe pour les macros, le compilateur peut y effectuer tous les contrôles syntaxiques et sémantiques qu'il fait sur les fonctions ordinaires.

La portée d'une fonction en ligne est réduite au fichier où la fonction est définie. Par conséquent, de telles fonctions sont généralement écrites dans des fichiers en-tête (fichiers « `.h` »), qui doivent être inclus dans tous les fichiers comportant des appels de ces fonctions.

1.5 Valeurs par défaut des arguments des fonctions

Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut. Exemple :

```
void trier(void *table, int nbr, int taille = sizeof(void *), bool croissant = true);
```

Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants peuvent alors être omis (ainsi que les virgules correspondantes), les paramètres formels seront initialisés avec les valeurs par défaut. Exemples :

```
trier(t, n, sizeof(int), false);
trier(t, n, sizeof(int));        // croissant = true
trier(t, n);                     // taille = sizeof(void *), croissant = true
```

1.6 Surcharge des noms de fonctions

La *signature d'une fonction* est la suite des types de ses arguments formels (et quelques éléments supplémentaires, que nous verrons plus loin). Le type du résultat rendu par la fonction ne fait pas partie de sa signature.

La *surcharge des noms des fonctions* consiste en ceci : en C++ des fonctions différentes peuvent avoir le même nom, à la condition que leurs signatures soient assez différentes pour que, lors de chaque appel, le nombre et les types des arguments effectifs permettent de choisir sans ambiguïté la fonction à appeler. Exemple :

```
int puissance(int x, int n) {
    calcul de x^n avec x et n entiers
}
double puissance(double x, int n) {
    calcul de x^n avec x flottant et n entier
}
double puissance(double x, double y) {
    calcul de x^y avec x et y flottants
}
```

On voit sur cet exemple l'intérêt de la surcharge des noms des fonctions : la même notion abstraite « x à la puissance n » se traduit par des algorithmes très différents selon que n est entier ($x^n = x.x...x$) ou réel ($x^n = e^{n \log x}$) ; de plus, pour n entier, si on veut que x^n soit entier lorsque x est entier, on doit écrire deux fonctions distinctes, une pour x entier et une pour x réel.

Or le programmeur n'aura qu'un nom à connaître, `puissance`. Il écrira dans tous les cas

```
c = puissance(a, b);
```

le compilateur se chargeant de choisir la fonction la plus adaptée, selon les types de `a` et `b`.

NOTES. 1. Le type du résultat rendu par la fonction ne fait pas partie de la signature. Par conséquent, on ne peut pas donner le même nom à deux fonctions qui ne différent que par le type du résultat qu'elles rendent.

2. Lors de l'appel d'une fonction surchargée, la fonction effectivement appelée est celle dont la signature correspond avec les types des arguments effectifs de l'appel. S'il y a une correspondance exacte, pas de problème. S'il n'y a pas de correspondance exacte, alors des règles complexes (trop complexes pur les expliquer ici) s'appliquent, pour déterminer la fonction à appeler. Malgré ces règles, il existe de nombreux cas de figure ambigus, que le compilateur ne peut pas résoudre.

Par exemple, si `x` est flottant et `n` entier, l'appel `puissance(n, x)` est erroné, alors qu'il aurait été correct s'il y avait eu une seule définition de la fonction `puissance` (les conversions entier $\leftrightarrow$ flottant nécessaires auraient alors été faites).

1.7 Appel et définition de fonctions écrites en C

Pour que la compilation et l'édition de liens d'un programme avec des fonctions surchargées soient possibles, le compilateur C++ doit fabriquer pour chaque fonction un *nom long* comprenant le nom de la fonction et une représentation codée de sa signature ; c'est ce nom long qui est communiqué à l'éditeur de liens.

Or, les fonctions produites par un compilateur C n'ont pas de tels noms longs ; si on ne prend pas de disposition particulière, il sera donc impossible d'appeler dans un programme C++ une fonction écrite en C, ou réciproquement. On remédie à cette impossibilité par l'utilisation de la déclaration « extern "C" » :

```
extern "C" {
    déclarations et définitions d'éléments
    dont le nom est représenté à la manière de C
}
```

Ceci est un exemple, cliquez sur le lien de téléchargement pour obtenir le cours complet.