Chapitre 1 : syntaxe, compilation

Quelques dates :
— 1972 : conception du langage `C`, encore un des langages de programmation les plus utilisés aujourd'hui.
— 1983 : conception du langage `C++`, sorte de "modernisation" du `C`.
— 1998 : première normalisation du `C++`.
— 2011 : dernière modification majeure du standard du `C++`.

1 Structure d'un programme

Un programme en `C++` est essentiellement une suite de *fonctions*. La syntaxe d'une fonction est la suivante :

```
type_retour F(args){
// Code de la fonction
}
```

Ici, `F` désigne le nom de la fonction, qui prendra des arguments listés dans `args`, et qui renvoiera une valeur de type `type_retour` (voir partie suivante).

Les sauts de lignes sont facultatifs, mais conseillés pour avoir un code lisible.

Le code de la fonctions sera composé d'une suite d'*instructions*, à savoir des appels de fonctions et des opération effectuées sur des variables. Toutes les fonction et variables utilisées doivent avoir été préalablement *déclarées* (voir partie suivante).

On sortira de la fonction dès que l'on rencontrera le mot-clé `return`, qui permettra aussi de préciser la valeur à renvoyer par la fonction.

Si on veut se servir d'une fonction avant d'écrire son code, il faut préalablement la déclarer par la syntaxe suivante :

```
type_retour F(args);
```

Cette déclaration est appelée le *prototype* de la fonction.

2 Les types

Un ordinateur ne sait manipuler que des suites de zéros et de uns (appelés *bits*). On va donc vouloir chercher à représenter des objets mathématiques par des suites de bits. La notion de *type* permet de savoir comment interpréter une suite de bits donnée.

Une instruction de la forme

```
type_var nom_var;
```

appelée *déclaration*, a pour effet de signaler au compilateur que `nom_var` est le nom d'une variable de type `type_var`. L'indication de `type_var` permet de savoir comment on doit comprendre les opérations qui vont être effectuées sur `nom_var`.

On présente maintenant les types présent par défaut.

2.1 Les entiers

Informatiquement, les entiers peuvent être représentés exactement : la donnée de n bit permet de représenter tous les entiers de 0 à $2^n - 1$ (ou de -2^{n-1} à $2^{n-1} - 1$) par le biais des chiffres binaire de l'entier. Les opérations effectuées sur les entiers seront en fait effectuées modulo 2^n.

Les types entiers sont, dans l'ordre du plus petit au plus grand nombre de bits, `char`, `short`, `int`, `long` et `long long`. Tous ces types existent en version `signed` (possibilité de coder des entiers négatifs) et `unsigned` (tous les entiers sont positifs). De base, un type entier est signé.

Le type `char` permet aussi de représenter 128 caractères par le biais de leur code ASCII.

2.2 Les nombres à virgule flottante

Comme une variable donnée ne peut prendre qu'un nombre fini de valeurs différentes (puisqu'elle est représentée sur un nombre de bits fixé), il est impossible de représenter n'importe quel réel.

Toutefois, tout réel non nul peut se mettre sous la forme $\pm(1+x)\times 2^n$, où x est un réel de $[0,1[$ et n est un entier relatif. Par exemple, on écrit $5 = 1.25 \times 2^2$, $0.1 = 1.6 \times 2^{-4}$ ou encore $\pi = 1.570796... \times 2$. On peut donc définir certaines valeurs réelles par la donnée :

— du signe du réel considéré, qui peut être codé sur un seul bit ;
— de la puissance de 2 nécessaire pour ramener le réel dans l'intervalle $[1,2[$ (le n de l'expression $\pm(1+x)\times 2^n$). Il s'agit d'un entier que l'on peut coder sur un certain nombre de bits.
— des chiffres situés après la virgule du réel ramené dans $[1,2[$. Il s'agit des chiffres du développement binaire de x dans $\pm(1+x)\times 2^n$.

Les types réels sont, du moins précis au plus précis, `float`, `double` et `long double`. Les opération sur les nombres à virgule flottante peuvent présenter des erreurs d'arrondi. On conseille d'utiliser le type `double`, dont les erreurs d'arrondi sont de l'ordre de 10^{-16} (en valeur relative).

2.3 Les booléens

Le type `bool` permet de représenter les deux valeurs "vrai" et "faux", notées `true` et `false`.

En fait, par convention, toute valeur nulle pourra jouer le rôle de `false` et toute valeur non-nulle pourra jouer le rôle de `true`.

3 Boucles

La plupart des langages de programmation permettent d'utiliser des boucles pour répéter des instructions un certain nombre de fois. Voici leur syntaxe en `C++`.

3.1 Les boucles `if`

La boucle `if` utilise la syntaxe suivante :

```
if (condition)
{
// Lu si condition est different de 0
}
else
{
```

```
// Lu si condition vaut 0
}
```

3.2 Les boucles `while`

La syntaxe

```
while (condition)
{
// Instructions
}
```

permet de répeter les instructions tant que `condition` est vraie.

Si l'on veut s'assurer que les instructions sont effectuées au moins une fois, on utilise la syntaxe :

```
do
{
// Instructions
}
while (condition)
```

3.3 Les boucles `for`

En `C++`, la boucle `for` est en fait une généralisation des boucles `if` et `while`. Précisément, la syntaxe est :

```
for (I1 ; I2 ; I3)
{
// Instructions
}
```

Ici, `I1`, `I2` et `I3` sont trois instructions. Au début de la boucle, l'instruction `I1` est effectuée. Ensuite, `I2` est évaluée, et si elle prend la valeur vrai, le contenu de la boucle est effectué. À la fin de la boucle, `I3` est évalué, puis on recommence : si `I2` est vraie, on repasse dans la boucle, puis on effectue `I3`, etc.

Cette syntaxe permet en fait d'écrire des boucles `for` ayant le même effet que des boucles `while` ou des boucles `if`. Ce n'est toutefois pas conseillé pour des raisons de lisibilité.

4 Gestion des entrées/sorties

Un programme sera bien sûr inutile si il ne peut pas renvoyer le résultat de son calcul. Il doit donc être possible de communiquer avec l'utilisateur par le biais des entrée (l'utilisateur fournit des paramètres/données au programme) et des sorties (le programme envoie à l'utilisateur des informations).

Les outils d'entrées/sorties standards sont inclus dans la bilitothèque `iostream` qui définit notamment deux flux : le flux d'entrée `std::cin` et le flux de sortie `std::cout`. Pour envoyer des informations dans ces flux, on utilise les opérateurs d'*injection* `<<` (pour envoyer des informations à afficher dans un flux de sortie) et d'*extraction* `>>` (pour recueillir des données depuis l'utilisateur ou depuis un fichier sur le disque). Par exemple la syntaxe :

```
std::cin >> n ;
std::out << n;
```

va stocker dans la variable `n` une valeur fournie par l'utilisateur, puis affichera cette valeur à l'écran.

Il est également possible d'envoyer ou de lire des données dans un fichier. Pour cela, on utilise la bibliothèque `fstream`.

5 Commentaires

Les commentaires en `C++` sont signalés par la syntaxe `//` : tout ce qui est écrit entre `//` et le saut de ligne suivant sera ignoré par le compilateur. Une autre syntaxe est `/*` et `*/` : tout ce qui est situé entre deux telles bornes est ignoré.

Note : il faut *toujours* commenter ses programmes. Cela permet d'économiser beaucoup de temps lors d'une consultation ultérieure.

6 Compilation

La *compilation* d'un programme consiste à créer un fichier exécutable à partir du code source. Dans ce cours, on utilisera le compilateur `g++`, présent par défaut sur toutes les distributions Linux.

La compilation d'un programme se déroule suivant les étapes suivantes :

— la *précompilation*, qui consiste à modifier le code source en suivant les instructions commençant par `#`. Cette étape nous servira quasi-exclusivement à inclure des bibliothèques avec l'instruction `#include`.

— La *compilation* à proprement parler, qui consiste à traduire le code `C++` en langage assembleur (une version du langage machine lisible par l'utilisateur).

— L'*assemblage* qui transforme le code assembleur en langage machine. Le résultat de cette étape est un *fichier objet*, signalé par l'extension `.o`.

— L'*édition de liens* : cette dernière phase consiste à rassembler les fichiers objets en un seul fichier executable.

En pratique on a deux situations possibles :

— Si notre programme est composé d'un unique fichier `prog.cpp`, on peut alors effectuer toutes les étapes de compilation d'un seul coup, à l'aide de la commande "`g++ prog.cpp`". Cette commande va créer dans le répertoire courant un ficher exécutable `a.out`, à exécuter en tapant "`./a.out`". Pour changer le nom par défaut du fichier exécutable, on emploie la syntaxe "`g++ prog.cpp -o prog`", qui crée un fichier exécutable nommé `propg`.

— Si le programme est composé de plusieurs fichiers, par exemple `fichier1.cpp` et `fichier2.cpp`, il faudra d'abord créer un fichier objet pour chaque fichier source avec "`g++ -c fichier1.cpp`" et "`g++ -c fichier2.cpp`". On obtiendra deux fichiers objets `fichier1.o` et `fichier2.o`. La commande "`g++ fichier1.o fichier2.o -o prog`" crée alors un exécutable `prog`.

Une étape inévitable dans la programmation est le fait de se tromper : on n'écrit jamais un programme correct du premier coup. Quand la syntaxe du programme est incorrecte, le compilateur va renvoyer un message d'erreur ("error"), ou un avertissement ("warning"). Les erreurs empêchent de créer un exécutable, alors que l'avertissement signale une syntaxe correcte, mais inhabituelle, qui pourrait avoir un effet différent de celui escompté.

Par défaut, `g++` ne signale que les erreurs, et pas les avertissement. Pour forcer l'affichage des avertissements, on utilisera l'option `-Wall` (`W` pour "warning", et `all` pour "tous") quand on utilisera `g++`. Par ailleurs, `g++` signale le numéro de la ligne où une erreur a été repérée.

Très important : il faut *toujours* lire les messages d'erreur !