Chapitre I : Les bases du C++

Le langage C++ est un langage de programmation puissant, polyvalent, on serait presque tenté de dire universel, massivement utilisé dans l'industrie du logiciel, et ce depuis de nombreuses années. On retrouve le C++ sur de nombreuses plates-formes, pour des types d'applications très diverses. Avec une syntaxe pouvant paraître difficile au premier abord, le C++ est rarement le premier langage de programmation que l'on apprend. C'est pourtant un langage simple, puissant, avec lequel on peut découvrir la programmation, apprendre de nombreuses techniques (programmation objet, programmation générique, programmation système, etc...)

Bien que C++ ait été créé à partir du langage C, ce n'est pas vraiment le même langage, et l'apprentissage préalable de C n'est pas du tout obligatoire. Il me paraît même plus simple de découvrir C à partir de C++ que l'inverse. On considère en général C comme un sous-ensemble de C++.

Le propos de ce livre n'étant pas le langage C, toute la syntaxe, les mots-clés, les fonctionnalités décrites ici seront celles du C++. Le langage C en lui-même ne sera pas abordé.

Dans ce chapitre nous étudierons la syntaxe, les concepts, les principes de bases du langage C++. Des exercices seront proposés tout au long du chapitre pour aider le lecteur à se familiariser avec C++.

1. Structure d'un programme C++

Le C++ est un langage **compilé**, ce qui signifie que le résultat final du travail du programmeur est un fichier comportant des instructions en **langage machine**, spécifiques à un processeur.

Pour arriver au résultat final (le programme exécutable) il faut passer par un certain nombre d'étapes dans le développement (je ne parle pas ici des phases d'analyse, de conception des algorithmes, de définition des besoins, de rédaction du cahier des charges, qui sont bien évidemment indispensables) :

- écriture du code en C++ dans un ou plusieurs fichiers source
- compilation de chaque fichier C++ en code objet
- liaison des fichiers objet entre eux, ajout des bibliothèques pour construire un **exécutable**
- déboguage de l'exécutable ainsi produit avant finalisation du produit

Chacune de ces étapes fait appel à un outil différent :

- un **éditeur de texte** (*text editor*) pour l'écriture du code source
- un **compilateur** (*compiler*) pour la production du code objet
- un **éditeur de liens** (*linker*) pour la production de l'exécutable
- un **débogueur** (*debugger*) pour la mise au point

Ces différents outils sont parfois regroupés au sein d'un même programme, appelé EDI (environnement de développement intégré) ou en anglais IDE (*integrated development environment*). Bien que C++ soit disponible sur de nombreuses plates-formes (PC sous Windows ou Linux, Macintosh, stations Sun, PDA ou téléphones portables, supercalculateurs, serveurs IBM, etc...) chaque **compilateur** est spécifique puisqu'il doit traduire le code C++ en instructions pour un processeur. De même, l'éditeur de liens doit utiliser des bibliothèques spécifiques à un système d'exploitation (MS-Windows, Linux, MacOS, NeXT, etc...), et donc est spécifique à la plate-forme utilisée.

On trouve des outils de développement C++ payants, en général regroupés au sein d'EDI (citons par exemple Borland C++ Builder, Microsoft Visual Studio sur PC), et des outils gratuits (citons le compilateur `gcc`, entre autres). Le langage C++ étant normalisé, si le développeur respecte la norme C++, tout compilateur C++ doit pouvoir compiler le code source écrit.

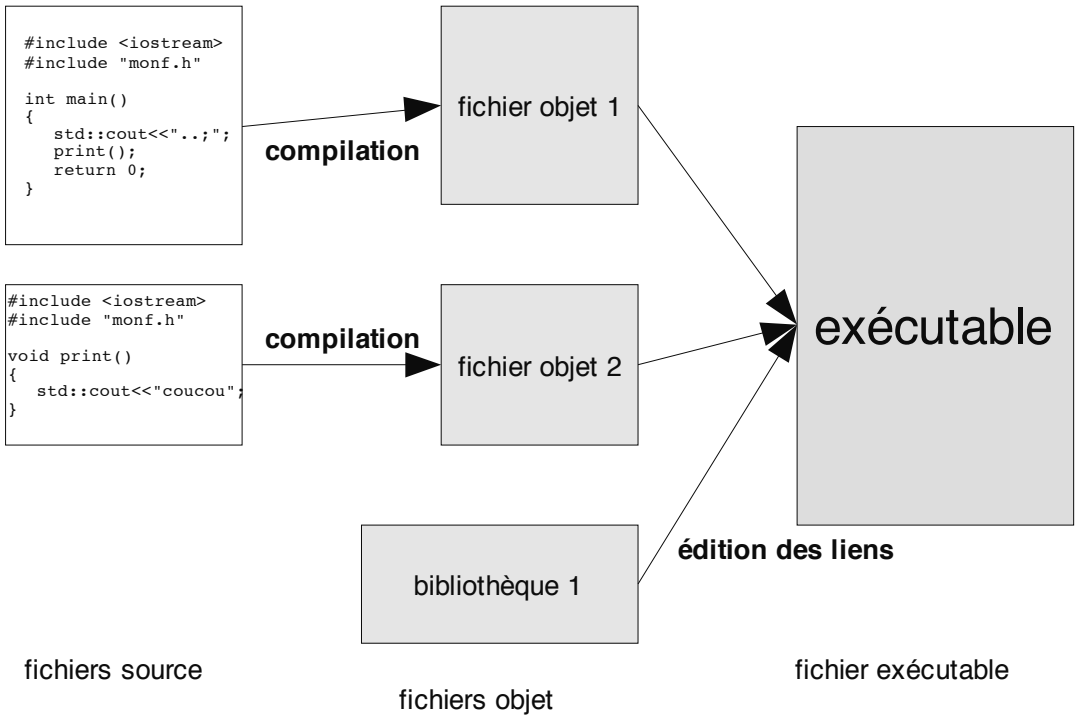

Fig 1 : schéma de principe d'un développement C++

On trouve donc un certain nombre de fichiers différents dans un développement C++ :

- des fichiers source, en mode texte (en général utilisation d'un codage `ASCII`), produits par un éditeur de texte et respectant la syntaxe C++. On les nomme souvent, par convention, en utilisant l'extension `cpp` (pour C plus plus).
- des fichiers source d'en-tête, qui ne sont pas compilés séparément mais intégrés avant compilation dans le fichier source, contenant des définitions, descriptions, et utilisant par convention l'extension `h` (pour *header*).
- des fichiers objet, en mode binaire, produits par le compilateur et respectant le codage des instructions du processeur cible (par exemple un *Intel Pentium* sur PC), dont l'extension est souvent `obj` ou `o`.
- un ou plusieurs fichiers exécutable(s), produits par l'éditeur de liens et pouvant être exécutés sur la plate-forme cible (par exemple un PC utilisant le système d'exploitation *Linux*).

Un fichier source C++ est appelé fréquemment **unité de compilation** puisque c'est l'entité de base vue par le compilateur. Un programme C++ pouvant être composé de plusieurs fichiers source différents (et compilés séparément) il est important de pouvoir faire communiquer entre elles ces unités de compilation. Cela implique un certain nombre de déclarations qu'il est fastidieux de répéter systématiquement, elles sont donc regroupées dans un fichier qu'il est possible d'intégrer au fichier source.

Pour utiliser par exemple, la fonction standard permettant de calculer une racine carrée, il faut indiquer au compilateur qu'il existe une fonction appelée `sqrt` . Or celle-ci est décrite dans des fichiers fournis par le compilateur (la bibliothèque standard, liée à votre code lors de l'édition des liens) : il faut donc indiquer au compilateur que cette fonction existe, dans une autre unité de compilation... Ceci peut se faire de la manière suivante :

```
namespace std
{
    extern double sqrt(double x);
}
```

Et à chaque fois qu'on désirera utiliser cette fonction dans une unité de compilation, ces lignes seront nécessaires !

Heureusement, ces déclarations sont réunies (ainsi que d'autres) dans des fichiers d'en-tête, qui ne sont pas des unités de compilation mais qui sont destinés à être intégrés dans votre fichier source. Pour cela, on utilise la directive `#include` suivie du nom du fichier d'en-tête à intégrer. Pour la fonction racine carrée, ce fichier est `cmath.` On trouvera donc, en début d'unité de compilation, le code :

```
#include <cmath>
```

`#include` n'est pas une instruction C++, mais une directive de pré processeur (un outil travaillant avant le compilateur sur votre fichier source).

Exercice 1. Compilation d'un source C++

⇨ À l'aide de votre EDI, ou d'un éditeur de texte simple, écrire un fichier texte contenant les lignes suivantes :

```
#include <iostream> // pour définir cout
int main()
{
    std::cout<<"Bonjour le C++"<<std::endl;
    return 0;
}
```

⇨ Sauver le fichier (par exemple sous *test1.cpp*).

⇨ Utiliser votre compilateur pour produire un fichier objet (par exemple *test1.obj*), puis l'éditeur de liens pour produire un exécutable (par exemple *test1.exe*).

⇨ Bien lire la documentation de votre outil de développement pour les éventuelles options de compilation.

2. Directives de préprocesseur

La plupart de ces directives sont spécifiques à un compilateur (voir donc la documentation de celui-ci) mais il en existe des standard.

#include permet d'inclure (insérer) un fichier texte dans le fichier source, en recopiant le contenu de celui-ci comme si vous l'aviez tapé. Le fichier peut se trouver dans un dossier spécifique au compilateur, dans le même dossier que le fichier source, ou dans un autre dossier. Suivant le cas, on utilise :

- `#include <fichier>` quand c'est un en-tête standard, stocké dans le dossier spécifique au compilateur.
- `#include "fichier"` quand le fichier se situe dans le même dossier que le fichier source.
- `#include "chemin\fichier"` ou `#include "chemin/fichier",` en précisant le chemin d'accès du fichier, si celui-ci se trouve dans un autre dossier.

`#define` permet de définir une **macro** qui sera étendue avant le traitement par le compilateur. Cette macro peut être simple (définition d'un alias par exemple) ou complexe (définition d'une macro-commande). Le langage C utilisait beaucoup

`#define` pour définir des pseudo fonctions génériques. **Ce n'est pas recommandé en C++**, où les **modèles** remplacent avantageusement cette procédure.

La syntaxe de `#define` est simple :

`#define` `<symbole à définir> [<valeur du symbole>]`, par exemple :

```
#define PI      3.1415 // utilise #define pour produire une constante
#define DEBUG    // définit une macro DEBUG sans valeur affectée
```

Il n'est pas conseillé d'utiliser cette directive pour définir une **constante** ou une macro-commande (comme cela se fait en C) : en C++ on utilisera **`const`** (voir I.5) pour définir une constante et les **modèles** (voir II.9) pour les macro-commandes génériques.

La principale utilisation de `#define` en C++ est, en utilisant également les directives `#ifdef`, `#ifndef`, de procéder à des **compilations conditionnelles**.

⇨ N'utilisez pas `#define` pour définir une constante symbolique.

⇨ N'utilisez pas `#define` pour définir une macro-commande.

`#ifdef, #ifndef, #if, #else et #endif` permettent de délimiter des blocs de compilation conditionnelle (c'est-à-dire un bloc de code à ne compiler que sous une certaine condition...

```
#if <condition>
// bloc de code compilé si la condition est vraie
#else
// bloc de code compilé si la condition est fausse
#endif
```

On peut ici remplacer `#if <condition>` par `#ifdef <symbole>` (condition vraie si un symbole est défini) ou `#ifndef <symbole>` (condition vraie si un symbole n'est pas défini). Ce mécanisme peut être très pratique dans divers cas :

- différents modes de compilation (par exemple : *debug* et *release*)
- différentes versions du même code
- travail en équipe
- protection contre les inclusions multiples d'en-tête.

Voici divers exemples d'utilisation de ces directives de compilation conditionnelle :

```
// cas 1 : plusieurs versions du même code
#if VERSION = 1
// code pour la version 1
#elseif VERSION=2
// code pour la version 2
#endif
```

```
// cas 2 : différents modes de compilation
#ifdef DEBUG
// code valable uniquement en mode debug
#endif

// code valable dans tous les modes

#ifndef DEBUG
// code valable uniquement en mode release
#endif
```

```
// cas 3 : protection contre les inclusions multiples
#ifndef CEFICHIER_H
#define CEFICHIER_H
// contenu du fichier...
#endif
// fin du fichier
```

⇨ Quand vous créez un fichier d'en-tête, utilisez les directives de compilation conditionnelles pour vous protéger des inclusions multiples.

Exercice 2. Directives de compilation

⇨ Créez un fichier `test.cpp` comportant le code suivant :

```
#include "test.h"
#include <iostream>
#include "test.h" // volontaire !

int main()
{
    std::cout<<"Bonjour les #include !"<<std::endl;
#ifdef TEST
    std::cout<<"Ceci n'apparait pas systématiquement";
#endif
    return 0;
}
```

⇨ Créez également un fichier `test.h` comportant le code suivant :

```
#ifndef TEST_H
#define TEST_H

#define TEST

#endif
```

⇨ Compilez `test.cpp` et créez un exécutable. Que s'affiche-t-il sur la console ? Que se passe-t-il si la première et la dernière directives du fichier `TEST.H` sont enlevées ?

⇨ Effacez la ligne `"#define TEST"` du fichier `TEST.H`, recompilez. Notez les modifications d'affichage.