Le programme "hello world"

Dans le chapitre précédent, vous avez vu la structure de base d'un programme C++ et un aperçu du processus de compilation. Cependant, le code présenté ne faisait rien, ce n'était pas très intéressant. Dans ce chapitre, vous allez voir comment afficher un message.

Pour illustrer cette fonctionnalité en C++, nous allons prendre le programme “hello world” comme exemple. Ce programme permet simplement d'afficher le message “hello, world!”. Il est traditionnellement utilisé pour montrer la syntaxe de base d'un langage informatique, ce qui explique qu'il possède son propre nom. Vous pouvez voir sur Wikipédia ce programme dans différents langages : Wikipédia.

Le programme *hello world* en C++ est assez proche du programme minimal présenté dans le chapitre précédent. Vous pouvez ouvrir ce code dans Coliru et copier-coller le code dans l'éditeur de votre choix.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Par rapport au code minimal, vous pouvez voir que l'on a ajouté deux lignes. La première ligne, qui contient la directive `#include`, permet de spécifier des fonctionnalités utilisées par le programme. “iostream” fournit les fonctionnalités de base pour afficher un message et récupérer les textes saisies par l'utilisateur.

La quatrième ligne, commençant par `std::cout`, permet l'affichage proprement dit. On voit sans problème le message à afficher “hello, world!” en clair dans le code. Vous pouvez vous amuser à changer le texte et voir ce que cela donne. L'instruction `std::endl` permet d'indiquer la fin d'une ligne et donc de passer à la ligne suivante (endl

pour “end line”, “fin de ligne”)

Ces deux lignes de code permettent d'utiliser le flux de sortie `std::cout` de la bibliothèque standard.

La bibliothèque standard

Lorsque l'on parle du C++, il faut en fait distinguer deux choses : le langage C++ et la bibliothèque standard. Le langage C++ proprement-dit propose uniquement les fonctionnalités fondamentales indispensables pour écrire un programme. La conséquence est que de nombreuses fonctionnalités ne sont pas intégrées directement dans le langage (même l'affichage d'un message ne fait pas partie du langage).

Heureusement, lorsqu'une fonctionnalité n'existe pas dans le langage, cela ne veut pas dire que l'on ne peut pas utiliser cette fonctionnalité. Il est possible d'écrire des bibliothèques, qui fournissent de nouvelles fonctionnalités utilisables en C++ (ou dans d'autre langages, mais cela sort du cadre de ce cours).

Cela veut dire aussi que si vous créez un programme qui propose des fonctionnalités intéressantes, vous pouvez également créer une bibliothèque pour que d'autres développeurs utilisent vos fonctionnalités (ou que vous puissiez vous même utiliser ces fonctionnalités dans plusieurs de vos programmes). La création d'une bibliothèque sera vue par la suite.

La bibliothèque standard fait partie intégrante du C++, il est indispensable d'apprendre à l'utiliser en même temps que le langage, l'un ne va pas sans l'autre.

Pour utiliser une fonctionnalité de la bibliothèque standard, il faut dans un premier temps le spécifier au compilateur, en utilisant la directive de pré-processeur `#include`. L'une des syntaxes de cette directive est la suivante (il en existe d'autres, mais qui ne seront pas utilisée avec la bibliothèque standard) :

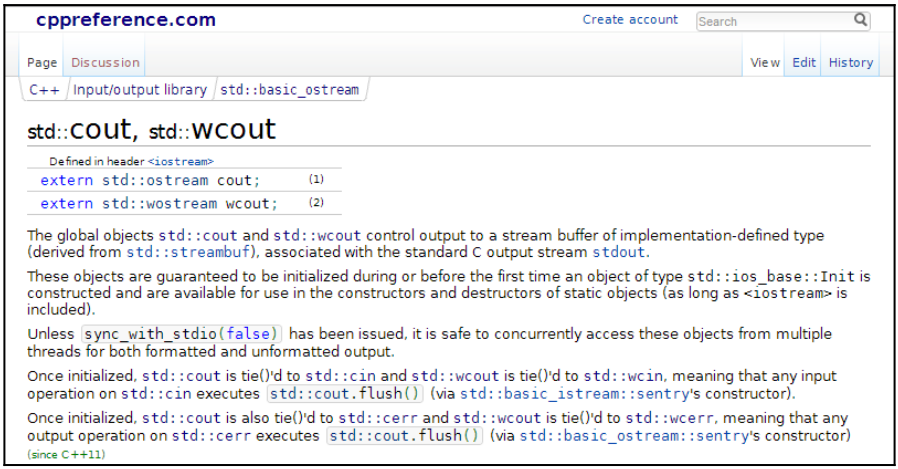
```
#include <nom_fichier>
```

Les directives de pré-processeur
Une directive de pré-processeur permet de paramétrer le comportement du pré-processeur lors de la compilation. Il existe différentes directives, vous en verrez plusieurs dans ce cours. Une directive s'écrit toujours avec un dièse suivi de la directive et d'éventuels paramètres optionnels.

Il n'est pas possible d'expliquer le fonctionnement de la directive `#include` sans expliquer avant le fonctionnement en détail du pré-processeur. Cela sera vu dans un chapitre dédié à la compilation en profondeur.

Elles sont parfois appellees “directive de compilation”.

Pour afficher un message, on utilise un objet particulier de la bibliothèque standard, nommé `std::cout`. Si vous regardez dans la documentation, vous voyez au début de la page qu'il est écrit : “Defined in header <iostream>”.



Cela vous indique quel fichier il faut inclure pour utiliser `std::cout` : le fichier `iostream` :

```
#include <iostream>
```

“iostream” correspond à ***I**nput/**O**utput stream*, ce qui signifie “flux

d'entrée et sortie". "Entrée" et "Sortie" doivent être compris du point de vue du programme : "entrée" d'information depuis l'extérieur vers l'intérieur du programme (par exemple saisie d'un texte par l'utilisateur ou la lecture d'un fichier) et "sortie" d'information depuis le programme vers l'extérieur (par exemple afficher un message à l'écran ou enregistrer dans un fichier). Vous verrez juste en dessous pourquoi on parle de "flux".

Lorsque vous utiliserez une fonctionnalité de la bibliothèque standard que vous ne connaissez pas, vous pourrez de la même manière aller rechercher dans la documentation quel fichier inclure.

L'espace de nom std

En C++, chaque chose doit avoir un nom unique, pour permettre au compilateur de les identifier correctement. Donner un nom n'est pas très compliqué, vous verrez par la suite les quelques règles à respecter. Lorsque l'on a un petit programme de quelques centaines ou milliers de ligne, cela pose pas trop de problème pour trouver des noms uniques. Mais dans le cas d'un programme de plus grande taille ou utilisant différentes bibliothèques, cela peut devenir très compliqué.

Pour éviter cette contrainte, le C++ permet de regrouper les noms dans un espace dédié : les espaces de noms (*namespace*). En créant un espace de noms, vous évitez les conflits entre les noms, ce qui peut simplifier vos codes. La bibliothèque standard utilise un espace de noms appelé `std`. Vous apprendrez par la suite à créer des espaces de noms, mais pour l'instant, voyons comment utiliser l'espace de noms de la bibliothèque standard.

Pour utiliser l'objet `cout` de la bibliothèque standard, il faut donc préciser que celui-ci provient de l'espace de noms `std`. Plusieurs solutions sont possibles, selon le contexte. Premièrement, vous pouvez déclarer l'espace de noms `std` à chaque utilisation d'une fonctionnalité de la bibliothèque standard, en utilisant l'opérateur `::` (comme vous l'avez vu dans les codes précédents) :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Dans ce cas, il faut faire précéder chaque utilisation de `cout` et de `endl` avec l'espace de noms.

La deuxième solution est de déclarer que vous allez utiliser une fonctionnalité d'un espace de noms en utilisant le mot-clé `using`. Lorsque le compilateur rencontre ensuite `cout`, il saura qu'il faut utiliser l'objet `std::cout` :

main.cpp

```cpp
#include <iostream>
using std::cout;

int main() {
    cout << "Hello, world!" << std::endl;
}
```

Vous pouvez remarquer ici que seul l'objet `cout` est déclaré en utilisant `using`. `endl` n'étant pas déclaré de cette manière, il faut l'écrire en utilisant la première syntaxe.

Pour terminer, il est possible d'activer un espace de noms globalement, en utilisant `using namespace`.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, world!" << endl;
}
```

Vous remarquez ici qu'il n'est plus nécessaire d'écrire `std::` devant `cout` et `endl` (ou n'importe quelle autre fonctionnalités de la bibliothèque

standard).

Cette syntaxe semble intéressante, puisque cela fait gagner du temps. Cependant, cela signifie que l'on perd l'intérêt des espaces de noms : si deux espaces de noms proposent le même identifiant, il y aura un conflit. Le message d'erreur généré par le compilateur ne sera pas forcement explicite, puisqu'il ne détectera pas le conflit d'espace de noms. Les erreurs générés seront du type “déclaration ambiguë” ou “déclaration multiple”.

Il est possible de limiter la portée de `using` en le déclarant à l'intérieur de la fonction :

main.cpp
```cpp
#include <iostream>

int main() {
    using std::cout;
    cout << "Hello, world!" << std::endl;
}
```

et :

main.cpp
```cpp
#include <iostream>

int main() {
    using namespace std;
    cout << "Hello, world!" << endl;
}
```

Il est préférable de limiter l'utilisation de la syntaxe avec `using namespace`. Dans ce cours, nous utiliserons systématiquement la première syntaxe, pour des raisons de compréhension. Dans vos codes, utilisez de préférence l'une des deux premières syntaxes.

Les flux standards

Si vous avez regardé un peu la documentation de `std::cout`, vous avez

peut-être remarqué qu'il existe d'autres flux de sortie :

- `cout` et `wcout` pour les messages standard ;
- `cerr` et `wcerr` pour les messages d'erreur ;
- `clog` et `wclog` pour les messages de log.

Par défaut, ces flux s'affichent tous dans le terminal, vous pouvez utiliser n'importe lequel. Le programme suivant :

main.cpp
```
#include <iostream>

int main() {
    std::cout << "le flux cout" << std::endl;
    std::wcout << "le flux wcout" << std::endl;
    std::cerr << "le flux cerr" << std::endl;
    std::wcerr << "le flux wcerr" << std::endl;
    std::clog << "le flux clog" << std::endl;
    std::wclog << "le flux wclog" << std::endl;
}
```

affiche :

```
le flux cout
le flux wcout
le flux cerr
le flux wcerr
le flux clog
le flux wclog
```

Il est possible de faire en sorte de récupérer spécifiquement un flux, par exemple pour l'enregistrer dans un fichier. Certains éditeur de code récupèrent par exemple les messages affichés à l'aide de `std::cerr` pour afficher les messages d'erreur.

Même si utiliser n'importe quel flux ne change rien à votre programme (si les flux ne sont pas spécifiquement récupérés), il est préférable de respecter le rôle de chaque flux et d'utiliser `std::cerr` pour les messages d'erreur, `std::clog` pour les messages d'information et `std::cout` pour les messages standards.

Internationalisation
Le `w` signifie que le flux prend en charge les caractères étendus (“w” pour *wide*), c'est-à-dire les caractères avec accent ou provenant d'un alphabet différent de l'anglais. La gestion de l’internationalisation est un peu complexe en C++, en particulier à cause de la multiplicité des normes de codage et la prise en charge très variable selon le système d'exploitation. Cela fera l'objet d'un chapitre dédié.

Si vous faites le test avec Clang dans Coliru, cela affichera les accents correctement, mais ça ne sera pas toujours le cas (en particulier sous Windows).

main.cpp
```
#include <iostream>

int main() {
    std::cout << "àâäéèêëîïôöùû" << std::endl;
}
```

Comment fonctionne un flux ? Imaginer un employé de bureau qui reçoit des dossiers. Il a une grande pile de dossiers, il prend le plus ancien, le traite, puis passe au suivant. Peu importe si les dossiers arrivent un par un ou en paquet, il les prend toujours un par un.

Les flux standards fonctionnent sur le même principe : ils reçoivent des données (les caractères à afficher), ils prennent le premier arrivé, l'affiche puis passent au suivant. L'opérateur permettant d'envoyer des données à un flux est l'opérateur `«` :

main.cpp
```
#include <iostream>

int main() {
    std::cout << "hello";
}
```

Ce code ne doit pas être compris comme signifiant “afficher 'hello'”, mais comme “envoyer les caractères 'h', 'e', 'l', 'l' et 'o' dans le flux standard 'std::cout'”.

Il est possible d'envoyer plusieurs valeurs en série de données dans un flux, en les séparant par plusieurs opérateurs `«` :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "hello" << "world";
}
```

Ce code signifie “envoyer 'hello' et 'world' dans 'std::cout'”, ce qui peut être développé en “envoyer les caractères 'h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd' dans 'std::cout'”. (Remarquez l'absence d'espace entre “hello” et “world”).

Il revient au même d'envoyer les données sur plusieurs lignes :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello";
    std::cout << "world";
}
```

Voire même d'envoyer les caractères un par un :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 'h';
    std::cout << 'e';
    std::cout << 'l';
    std::cout << 'l';
    std::cout << 'o';
    std::cout << 'w';
    std::cout << 'o';
    std::cout << 'r';
    std::cout << 'l';
    std::cout << 'd';
```

```
}
```

Tous les codes précédent affichent :

```
helloworld
```

Programmation impérative

Il existe plusieurs façons de concevoir un programme informatique, appelés paradigme de programmation. Le C++ est un langage qui autorise l'utilisation de plusieurs paradigme, il est multi-paradigme. Il supporte en particulier la programmation impérative, que va être détaillé dans la suite, la programmation générique et la programmation objet, qui seront vues plus tard dans ce cours, la programmation fonctionnelle, qui sera simplement citée, etc.

La programmation impérative décris un programme comme une suite d'instructions, qui modifie l'état du programme. Cette approche permet de suivre un programme pas-à-pas, depuis le début du programme (correspondant à la fonction `main` en C++) jusqu'à sa fin (lorsque la fonction `main` se termine en C++).

Les instructions sont exécutées une par une, dans l'ordre où elle apparaissent dans le code. Une instruction commence dès que la précédente se termine.

Vous pourrez également entendre parler de programmation procédurale et de programmation structurée, toutes deux appartenant au paradigme impératif. Dans la programmation procédurale, le programme est décomposé en “procédures” (en C++, on parle de “fonctions”) qui sont des suites d'instructions. Dans la programmation structurée, le programme n'est pas linéaire, mais ajoute les concepts de boucles (une suite d'instruction qui est répétée) et de conditions (une suite d'instruction qui peut être exécutée ou non). Cela sera détaillé dans la suite de ce cours.

Les ordinateurs modernes sont généralement capables d'exécuter plusieurs instructions en même temps, voire plusieurs programme en même temps. Cela complique forcement la compréhension du déroulement d'un programme. Pour simplifier les explications, ce cours se base sur une situation parfaite et abstraite, dans laquelle les instructions sont exécutées une par une.
Voyons sur un code simple comment suivre le déroulement d'un programme C++. Lorsque vous exécutez le programme suivant, que se passe-t-il ?

main.cpp
```cpp
#include <iostream>

int main() {
    std::cout << "ligne 1" << std::endl;
    std::cout << "ligne 2" << std::endl;
    std::cout << "ligne 3" << std::endl;
}
```

Pour commencer, le système lance le programme et appelle la fonction `main`. Dans cette étape, il se passe beaucoup de choses, mais qui concernent le système (et qui sont relativement complexes). Cela sort du cadre de ce cours. Mais vous pouvez considérer que le programme commence au niveau de la ligne contenant le `main`.

A cette étape, le programme n'a encore rien écrit dans la console (le système peut avoir écrit des choses, mais cela ne concerne pas le programme).

Une fois que le programme est lancé, la première instruction est exécutée. Cette première instruction est la première ligne contenant le `std::cout`, ce qui produit l'affichage de texte dans la console. La console affiche donc à la fin de cette étape :

```
ligne 1
```

L'étape suivante est la seconde ligne contenant `std::cout` et la console affiche donc :

```
ligne 1
ligne 2
```

La troisième étape est la ligne contenant le dernier `std::cout` :

```
ligne 1
ligne 2
ligne 3
```

Pour terminer, le programme arrive à la fin de la fonction `main`, correspondant à l'accolade fermante `}`. Le programme se termine et le système reprend la main.

Pour simplifier la lecture du code, chaque instruction est écrite sur une ligne, se terminant par un point-virgule `;`. Mais si une ligne contient plusieurs instructions, séparées par des points-virgules, cela ne change pas le déroulement du programme : chaque instruction est exécutée une par une. La présentation du code n'influence pas le déroulement du programme.
Pour suivre le déroulement d'un programme C++, vous avez deux garanties :

- chaque instruction est exécutée ;
- les instructions sont exécutées dans l'ordre.

Il n'est donc pas possible d'obtenir les résultats suivants dans la console :

```
ligne 1
ligne 3
```

ou

```
ligne 1
ligne 3
ligne 2
```

Le comportement d'un programme sera dont prédictible (sauf erreur de programmation) et constant.

Exercices

- Testez le code en “oubliant” de mettre la directive de compilation `#include`. Quels messages d'erreur sont produit ? Trouvez-vous que les messages sont explicites et permettent de trouver facilement le problème ?
- Testez le code en “oubliant” de mettre l'espace de noms `std` devant `cout` et `endl`. Même questions que précédemment.
- Testez d'autres erreurs dans le code et regardez les messages d'erreur produits.

Chapitre précédent	Sommaire principal	Chapitre suivant