

T7.2 Réaliser une maquette, un prototype logiciel/matériel

C4.4 Développer un module logiciel.

S4.7 Langages de programmation C++

*Table des matières*

1	Sites de référence .....	2
2	Utilisation d'un éditeur-compileur en ligne .....	2
3	Structure d'un programme .....	3
4	Commentaires.....	6
5	Utilisation du namespace std .....	6
6	Un peu d'anglais .....	7
7	Exercices .....	7

{ C / C++ }

# 1 SITES DE REFERENCE

<http://en.cppreference.com/w/>

site de référence pour le C++, mais aussi le C, en anglais

<http://fr.cppreference.com/w/>

le même en français, mais moins complet, et moins à jour, et mal traduit parfois

<https://isocpp.org/wiki/faq>

une faq (questions fréquemment posées), en anglais

[https://fr.wikibooks.org/wiki/Programmation\\_C%2B%2B](https://fr.wikibooks.org/wiki/Programmation_C%2B%2B)

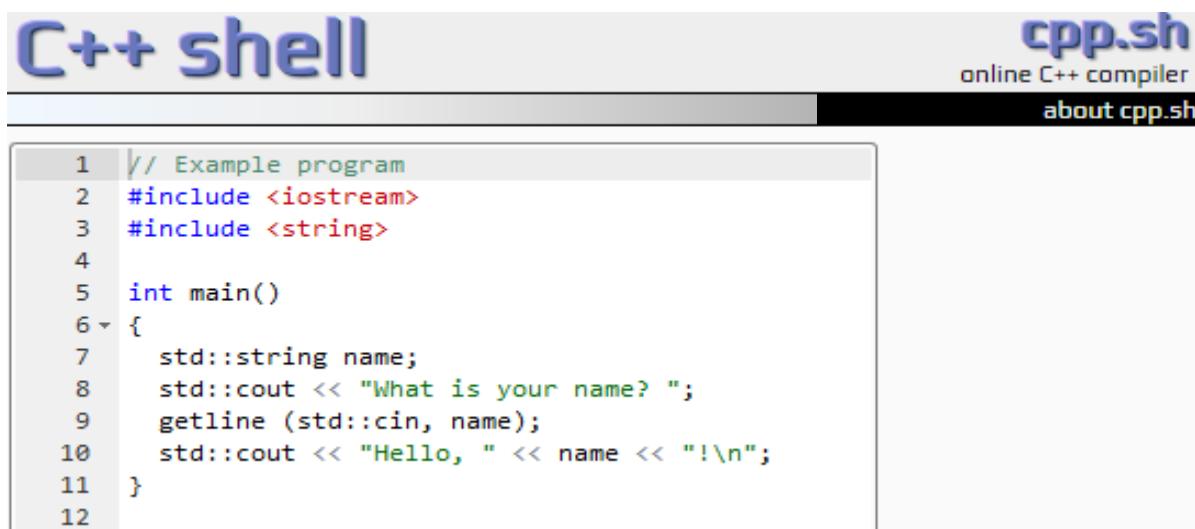
en français

# 2 UTILISATION D'UN EDITEUR-COMPILATEUR EN LIGNE

Pour saisir et tester les programmes des chapitres suivants nous allons utiliser un outil gratuit disponible sur internet à l'adresse <http://cpp.sh/>

En cas de problème vous pouvez aller sur un des sites suivant qui fournit les mêmes à peu près les mêmes services :

<https://repl.it/languages/cpp11> , ou <http://melpon.org/wandbox/> .

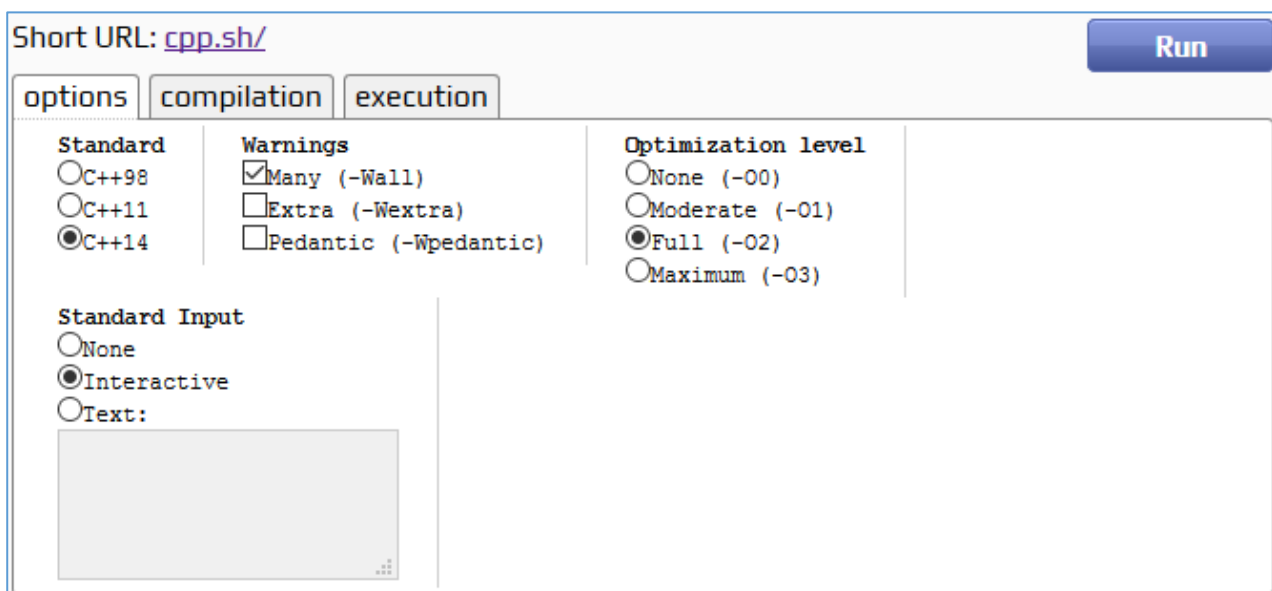


```
1 // Example program
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string name;
8     std::cout << "What is your name? ";
9     getline (std::cin, name);
10    std::cout << "Hello, " << name << "!\n";
11 }
12
```

Il faut d'abord effacer le programme exemple qui apparaît, puis saisir son propre programme en fonction des exercices.

Dans la partie inférieure il y a trois onglets :

- L'onglet « options » que l'on configure comme ci-dessous (options expliquées dans un prochain cours)
- L'onglet « compilation » (y apparaîtront des messages d'erreurs/d'avertissement si programme mal saisi)
- L'onglet « exécution » (affiche les sorties et résultats du programme, qu'on lance en cliquant sur « Run »)



Short URL: [cpp.sh/](http://cpp.sh/) Run

**options** **compilation** **execution**

<b>Standard</b> <input type="radio"/> C++98 <input type="radio"/> C++11 <input checked="" type="radio"/> C++14	<b>Warnings</b> <input checked="" type="checkbox"/> Many (-Wall) <input type="checkbox"/> Extra (-Wextra) <input type="checkbox"/> Pedantic (-Wpedantic)	<b>Optimization level</b> <input type="radio"/> None (-O0) <input type="radio"/> Moderate (-O1) <input checked="" type="radio"/> Full (-O2) <input type="radio"/> Maximum (-O3)
---	---	---

**Standard Input**  
☐ None  
☒ Interactive  
☐ Text:

### 3 STRUCTURE D'UN PROGRAMME

La meilleure façon d'apprendre un langage de programmation est l'écriture de programmes.

En règle générale, le premier programme pour débutants est un programme appelé "Hello World", qui se contente d'écrire "Hello World" à l'écran de votre ordinateur.

Bien qu'il soit très simple, il contient tous les composants fondamentaux d'un programme C ++ :

```
1 // Mon premier programme en C++
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Hello World!";
7 }
```

Hello World!

Le panneau de gauche ci-dessus montre le code C ++ pour ce programme. Le panneau de droite montre le résultat lorsque le programme est exécuté par un ordinateur. Les numéros gris à gauche des panneaux sont des numéros de ligne pour pouvoir parler du programme et rendre la recherche d'erreurs plus facile. Ils ne font pas partie du programme.

Examinons ce programme ligne par ligne :

**Ligne 1 :** `// Mon premier programme en C++`

Deux signes slash indiquent que le reste de la ligne est un commentaire inséré par le programmeur, mais qui n'a aucun effet sur le comportement du programme. Les programmeurs les utilisent pour inclure des explications ou des observations courtes concernant le code ou le programme. Dans ce cas, il est une brève description du programme.

**Ligne 2 :** `#include <iostream>`

Les lignes commençant par un signe dièse (#) sont des directives lues et interprétées par ce qui est appelé le *préprocesseur*. Ce sont des lignes spéciales interprétées avant la compilation proprement dite du programme. Dans ce cas, la directive `#include <iostream>`, indique au préprocesseur d'inclure une section de code C ++ standard, connu sous le nom en-tête *iostream*, qui permet d'effectuer des opérations d'entrée et de sortie standard, telles que l'écriture de la sortie de ce programme (Hello World) à l'écran.

**Ligne 3 :** Une ligne vide.

Les lignes vides n'ont aucun effet sur un programme. Elles améliorent simplement la lisibilité du code.

**Ligne 4 :** `int main ()`

Cette ligne initie la déclaration d'une fonction. En gros, une fonction est un groupe d'instructions de code, groupe auquel on a donné un nom : dans ce cas, cela donne le nom « main » au groupe des instructions de code qui suivent. Les fonctions seront étudiées en détail dans un chapitre ultérieur.

La fonction nommée `main` est une fonction spéciale dans tous les programmes C ++; c'est la fonction appelée lorsque le programme est lancé. L'exécution de tous les programmes C ++ commence par la fonction `main` (peu importe où la fonction est située dans le reste du code).

## Lignes 5 et 7 : {et}

L'accolade ouverte { à la ligne 5 indique le début de la définition de la fonction `main`, et l'accolade de fermeture } à la ligne 7, indique sa fin. Tout ce qui est entre ces accolades est le corps de la fonction, et définit ce qui se passe quand `main` est appelée. Toutes les fonctions utilisent des accolades pour indiquer le début et la fin de leurs définitions.

## Ligne 6 : `std::cout << "Hello World!";`

Cette ligne est une instruction C++. Une instruction est une expression qui fait une action. C'est le cœur d'un programme, la partie qui décrit son comportement réel. Les instructions sont exécutées dans l'ordre dans lequel elles apparaissent dans le corps d'une fonction.

Cette instruction comporte trois parties : d'abord, `std::cout`, qui identifie le « **standard character output** » (sortie standard, en général c'est l'écran d'ordinateur). En second lieu, l'opérateur d'insertion ( `<<` ), qui indique que ce qui suit est inséré dans `std::cout`. Enfin, une phrase entre guillemets ("Hello World !"), qui est le contenu inséré dans la sortie standard.

Notez que l'instruction se termine par un point - virgule (;). Ce caractère marque la fin de l'instruction. Toutes les instructions de C++ doivent se terminer par un point-virgule. L'une des erreurs de syntaxe les plus courantes dans C++ est d'oublier de mettre fin à une instruction avec un point-virgule.

Vous avez peut-être remarqué que toutes les lignes de ce programme n'effectuent pas des actions lorsque le code est exécuté. Il y a une ligne contenant un commentaire (commençant par //). Il y a une ligne avec une directive pour le préprocesseur (commençant par #). Il y a une ligne qui définit une fonction (dans ce cas, la fonction `main`). Et, enfin, une ligne avec des instructions se terminant par un point-virgule (l'insertion dans `cout`), qui était dans le bloc délimité par les accolades ( { } ) de la fonction `main`.

Le programme a été structuré en différentes lignes et avec des indentations correctes, afin de le rendre plus facile à lire et à comprendre. Mais C++ n'a pas de règles strictes sur l'indentation ou sur la façon de diviser les instructions dans les différentes lignes.

Par exemple, au lieu de :

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

Nous aurions pu écrire :

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

Le tout sur une seule ligne, ce qui aurait eu exactement le même sens que le code précédent.

En C++, la séparation entre les instructions est indiquée par un point-virgule à la fin (;), et la séparation en différentes lignes n'est pas obligatoire. De nombreuses instructions peuvent être écrites sur une seule ligne, ou chaque instruction peut être sur sa propre ligne. La division de code sur différentes lignes ne sert qu'à le rendre plus lisible pour les personnes qui veulent le lire, mais n'a aucun effet sur le comportement réel du programme.

Maintenant, nous allons ajouter une instruction supplémentaire à notre premier programme :

```
// Mon second programme en C++
#include <iostream>

int main ()
{
    std::cout << "Hello World! ";
    std::cout << "Je suis un programme C++";
}
```

Hello World! Je suis un programme C++

Dans ce cas, le programme effectue deux insertions dans `std::cout`, dans deux instructions différents. Encore une fois, la séparation dans différentes lignes de code donne simplement une meilleure lisibilité du programme, puisque `main` aurait pu être parfaitement valide définie d'une des 2 manières suivantes :

```
int main ()
{
    std::cout <<
        "Hello World! ";
    std::cout <<
        "Je suis un programme C++";
}
```

```
int main (){ std::cout << "Hello World! "; std::cout << "Je suis un programme C++"; }
```

Les directives de préprocesseur (celles qui commencent par #) sont en dehors de cette règle générale, car elles ne sont pas des instructions. Ce sont des lignes lues et traitées par le préprocesseur avant le début de la compilation proprement dite. Les directives de préprocesseur doivent être spécifiées sur leur propre ligne et, parce qu'elles ne sont pas des instructions, ne doivent **pas** se terminer par un point - virgule ( ; ).

Essayer maintenant ceci :

```
// Mon second programme en C++
#include <iostream>

int main ()
{
    std::cout << "Salut la foule! "
    std::cout << std::endl ;
    std::cout << "mon premier programme C++";
}
```

Puis ceci :

```
// Mon second programme en C++
#include <iostream>

int main ()
{
    std::cout << "Salut la foule! " << std::endl ;
    std::cout << "mon premier programme C++";
}
```

On obtient ce résultat, avec le texte affiché sur 2 lignes :

```
Salut la foule!
mon premier programme C++
```

## 4 COMMENTAIRES

Comme indiqué ci-dessus, les commentaires n'affectent pas le fonctionnement du programme ; cependant, ils constituent un outil important pour documenter directement dans le code source ce que fait le programme, et comment il fonctionne.

C++ prend en charge deux manières de commenter le code :

```
// Commentaire de ligne
/* commentaire de bloc */
```

Le premier d'entre eux, connu en tant que *commentaire de ligne*, fait que tout ce qui se trouve entre la paire de signes slash ( // ) et à la fin de cette même ligne, est du commentaire. Le second, connu sous le nom de *commentaire de bloc*, transforme en commentaire tout ce qui se trouve entre les caractères /\* et la première apparition des caractères \*/, avec la possibilité d'inclure plusieurs lignes.

Nous allons ajouter des commentaires à notre deuxième programme :

```
/* Mon second programme en C++
   avec plus de commentaires */

#include <iostream>

int main ()
{
    std::cout << "Salut la foule! " << std::endl; // affiche: Salut la foule!

    std::cout << "mon premier programme C++";      // affiche: mon premier programme C++
}
```

Si des commentaires sont inclus dans le code source d'un programme sans utiliser les caractères de commentaire ( // ou alors /\* et \*/ ), le compilateur les prend comme s'ils étaient des instructions de C++ , et donc probablement en causant l' échec de la compilation avec un ou plusieurs messages d'erreur.

## 5 UTILISATION DU NAMESPACE STD

Si vous avez vu du code C++ avant, vous avez peut - être vu `cout` utilisé au lieu de `std::cout` . Les deux nomment le même objet : le premier utilise son *nom non qualifié* ( `cout` ), tandis que le second le *qualifie* directement dans *l'espace de noms std* (comme `std::cout` ).

`cout` fait partie de la bibliothèque standard, et tous les éléments de la bibliothèque standard C++ sont déclarés au sein de ce que l'on a appelé un *espace de noms*: l'espace de noms `std` .

Afin de se référer aux éléments dans l'espace de nom `std` un programme doit soit se qualifier à chaque utilisation d'éléments de la bibliothèque (comme nous l'avons fait en préfixant `cout` avec `std::` ), ou introduire une visibilité de ses composants. La façon la plus typique de permettre la visibilité de ces composants est d'utiliser une déclaration de type *using* :

```
using namespace std;
```

La déclaration ci-dessus permet à tous les éléments de l'espace de noms `std` d'être accessibles d'une manière *non qualifiée* (sans le préfixe `std::` ).

Dans cet esprit, le dernier exemple peut être réécrit pour faire des usages non qualifiés de `cout` comme :

```
// Mon second programme en C++
#include <iostream>
using namespace std;

int main ()
{
    cout << "Hello World! ";
    cout << "I'm a C++ program";
}
```

Les deux manières d'accéder aux éléments de l'espace de noms `std` (qualification explicite ou avec `using`) sont valables en C++ et produisent le même comportement.

Par souci de simplicité, et pour améliorer la lisibilité, les exemples de ces cours seront le plus souvent utilisés avec cette dernière approche, donc avec l'utilisation de **using**.

Mais il faut bien noter que la *qualification explicite* (avec le préfixe `std::`) est la seule façon de garantir que des conflits de noms ne se produisent jamais.

Les espaces de noms sont expliqués plus en détail dans un chapitre ultérieur.

## 6 UN PEU D'ANGLAIS

Espaces de noms	namespaces
Dièse #	hash
Accolades { et }	braces
Point virgule ;	semicolon
Deux points :	colon

## 7 EXERCICES

1. Relire le cours (10min) pour réviser ce qui a été vu et mémoriser
2. Fermer le cours pour essayer de faire l'exercice de mémoire
3. Partir d'une page vierge du site internet <http://cpp.sh/> ou un des autres indiqués.
4. Ecrire un programme qui affiche :

options	compilation	execution	
Bonjour les BTS SN-IR Je m'appelle Prénom NOM			

5. N'oubliez pas de commenter votre en-tête de programme (date, nom de l'auteur, description de ce que fait le programme)
6. Commenter aussi vos lignes de programme