Présentation du langage C++

IUT Lyon 1

Yosra ZGUIRA

yosra.zguira@insa-lyon.fr

2016 - 2017

Introduction

- **Le langage C++** a été conçu à partir de 1982 par Bjarne Stroustrup (AT&T Bell Laboratories), comme une extension du langage C, lui-même créé dès 1972 par Denis Ritchie, formalisé par Kerninghan et Ritchie en 1978.

- L'objectif principal de B. Stroustrup était d'ajouter des classes au langage C et donc, en quelque sorte, de « greffer » sur un langage de programmation procédurale classique des possibilités de « **programmation orientée objet** »(en abrégé P.O.O.).

- Après 1982, les deux langages C et C++ ont continué d'évoluer parallèlement.

  ✓ C a été normalisé par l'ANSI (American National Standards Institute) en 1990.

  ✓ C++ a connu plusieurs versions, jusqu'à sa normalisation par l'ANSI en 1998.

Présentation du C++

- Le C++ est un langage **multiparadigme**. Il supporte essentiellement les paradigmes suivants :

  ✓ **Programmation procédurale**: il reprend essentiellement les concepts du langage C, notamment la notion de fonction (une procédure étant une fonction avec un retour de type 'void').

  ✓ **Programmation structurée:** il reprend la notion **struct** du langage C. Cette notion est considérée en C++ aussi comme des classes dont l'accès par défaut est public.

  ✓ **Programmation orientée-objet**: il implémente la notion de **classe**, **d'encapsulation**, **d'héritage** (simple ou multiple), **d'abstraction** grâce aux classes de base abstraites pures, de **polymorphisme dynamique** grâce aux fonctions membres virtuelles.

  ✓ **Programmation générique**: il implémente les fonctions et les classes génériques.

- C++ et la programmation structurée

- C++ et la programmation orientée objet

- C et C++

- C++ et la bibliothèque standard

- Les forces de C++

La programmation structurée

- Un programme peut comporter des milliers, et même des millions d'instructions.
- Impossible pour un développeur d'avoir une vision d'ensemble et de gérer cette complexité.

  ➔ Il faut décomposer le programme en éléments plus simples à appréhender.

- Une mauvaise décomposition peut rendre le problème encore plus difficile à résoudre.

- La programmation structurée permet de bien décomposer un programme complexe en parties plus simples à comprendre.

- Un programme est décomposé en modules.

- Chaque module a une des structures de la programmation structurée.

Programmes = algorithmes + structures de données

C++ et la programmation structurée

- Les possibilités de programmation structurée de C++ sont en fait celles du langage C et sont assez proches de celles des autres langages, à l'exception des pointeurs.

- La notion de procédure se retrouvera en C++ dans la notion de fonction.

- La transmissions des arguments pourra s'y faire, au choix du programmeur **: par valeur**, **par référence** (ce qui n'était pas possible en C) ou encore par le biais de manipulation de **pointeurs**.

C++ et la programmation orienté objet

- Les possibilités de la programmation orienté objet (POO) représentent bien sûr l'essentiel de l'apport de C++ au langage C.

- C++ dispose de la notion de classe (généralisation de la notion de type défini par l'utilisateur).

- Une classe comportera :

  ✓ la description d'une structure de données ,

  ✓ des méthodes.

C et C++ (1/2)

- C++ est un « sur-ensemble » du langage C, offrant des possibilités de P.O.O.
- Certaines possibilités du C deviennent inutiles (ou redondantes) en C++.
- Par exemple, C++ a introduit de nouvelles possibilités d'**entrées-sorties** (basées sur la notion de flot) qui rendent superflues les fonctions standards de C telles que printf ou scanf.
- C++ dispose d'opérateurs de gestion dynamique (**new** et **delete**) qui remplacent avantageusement les fonctions malloc, calloc et free du C.

C et C++ (2/2)

- Par rapport au **langage C**, le **C++** apporte :

  - ✓ les **concepts orientés objet** (encapsulation, héritage) ;
  - ✓ les **références** ;
  - ✓ la **vérification stricte des types** ;
  - ✓ les **valeurs par défaut** des paramètres de fonctions ;
  - ✓ la **surcharge de fonctions**;
  - ✓ la **surcharge des opérateurs** (pour utiliser les opérateurs avec les objets) ;
  - ✓ les **templates** de classes et de fonctions ;
  - ✓ les **constantes typées** ;
  - ✓ la possibilité de **déclaration de variables** entre deux instructions d'un même bloc.

C++ et la bibliothèque standard

- Comme tout langage, C++ dispose d'une **bibliothèque standard** (fonctions et de classes prédéfinies).

- Elle comporte notamment de nombreux patrons de classes et de fonctions permettant de mettre en œuvre les structures de données les plus importantes (vecteurs dynamiques, listes chaînées, chaînes...) et les algorithmes les plus usuels.

- C++ dispose de la totalité de la bibliothèque standard du C, y compris de fonctions devenues inutiles ou redondantes.

Les forces de C++ (1/2)

- **Très répandu**

  ➔ Il fait partie des langages de programmation les plus utilisés sur la planète.

- **Rapide**

  ➔ Ce qui en fait un langage de choix pour les applications critiques qui ont besoin de performances. C'est en particulier le cas des jeux vidéo, les outils financiers ou de certains programmes militaires qui doivent fonctionner en temps réel.

- **Portable**

  ➔ Un même code source peut théoriquement être transformé sans problème en exécutable sous Windows, Mac OS et Linux. Vous n'aurez pas besoin de réécrire votre programme pour d'autres plates-formes !

Les forces de C++ (2/2)

- **Nombreuses bibliothèques** pour le C++.

➔ Les bibliothèques sont des extensions pour le langage, un peu comme des plug-ins. De base, le C++ ne sait pas faire grand chose mais, en le combinant avec de bonnes bibliothèques, on peut créer des programmes 3D, réseaux, audio, etc.

- **Multi-paradigmes**

➔ On peut programmer de différentes façons en C++ tel que par exemple la *Programmation Orientée Objet* (POO). C'est une technique qui permet de simplifier l'organisation du code dans nos programmes et de rendre facilement certains morceaux de codes réutilisables.

Les commentaires en C++

Les commentaires

❑ Commentaire de fin de ligne:

- Un commentaire de fin de ligne débute par un double slash **//** et se termine au prochain retour à la ligne.
- Exemple :

```
x++;   // augmenter x de 1
```

❑ Bloc de commentaires:

- Un bloc de commentaire est délimité par les signes slash-étoile /* et étoile-slash */ comme en Java.
- Exemple :

```
/* Un commentaire explicatif sur
plusieurs lignes... */
```

Les types de base de C++

Les variables (1/3)

- Comme la plupart des langages de programmation, le C++ utilise la notion de variable.

- Une variable peut être vue comme une zone de la mémoire qui comprend une certaine valeur.

- Les données manipulées en langage C++, comme en langage C, sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation.

Les variables (2/3)

- En langage C++, les noms de variables peuvent être aussi long que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères.

- Elles doivent répondre à certains critères :

  - ✓ un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)

  - ✓ un nom de variable peut comporter des lettres, des chiffres et le caractère « _ » (les espaces ne sont pas autorisés !)

  - ✓ les noms de variables ne peuvent pas être les noms suivants (qui sont des noms réservés) :

- asm, auto
- break
- case, catch, char, class, const, continue
- default, delete, do, double
- else, enum, extern
- float, for, friend
- goto
- if, inline, int
- long
- new
- operator
- private, protected, public
- register, return
- short, signed, sizeof, static, struct, switch
- template, this, throw, try, typedef
- union, unsigned
- virtual, void, volatile
- while

Les variables (3/3)

Nom de variable correct	Nom de variable incorrect	Raison
Variable	Nom de Variable	comporte des espaces
Nom_De_Variable	123Nom_De_Variable	commence par un chiffre
nom_de_variable	toto@mailcity.com	caractère spécial @
nom_de_variable_123	Nom-de-variable	signe - interdit
_707	this	nom réservé

Les noms de variables sont sensibles à la casse.

➔ Le langage C++ fait la différence entre un nom en majuscules et un nom en minuscules

La déclaration de variables

- Pour pouvoir utiliser une variable, il faut la définir.

  ➔ lui donner un nom et un type de donnée à stocker afin qu'un espace mémoire conforme au type de donnée qu'elle contient lui soit réservé.

- Une variable se déclare de la façon suivante:

```
type     Nom_de_la_variable;
```

- ou bien s'il y a plusieurs variables du même type:

```
type     Nom_de_la_variable1, Nom_de_la_variable2, ...;
```

Affectation d'une donnée à une variable

- Pour stocker une donnée dans une variable que l'on a initialisée, il faut faire une **affectation**, c'est-à-dire préciser la donnée qui va être stockée à l'emplacement mémoire qui a été réservé lors de l'initialisation.

- Pour cela on utilise l'opérateur d'affectation **« = »**:

```
Nom_de_la_variable  =  donnée;
```

- Pour stocker le caractère A dans la variable que l'on a appelée Caractere, il faudra écrire :

```
Caractere = 'A';
```

Initialisation d'une variable

- La déclaration d'une variable ne fait que « réserver » un emplacement mémoire où stocker la variable.

- Tant que l'on ne lui a pas affecté une donnée celle-ci contient ce qui se trouvait précédemment à cet emplacement, que l'on appelle ***garbage*** (en français *détritus*).

- On peut donc affecter une valeur initiale à la variable lors de sa déclaration, on parle alors d'initialisation :

```
type  Nom_de_la_variable = donnee;
```

- Exemple:

```
int   compteur  =  0;
```

Portée des variables (1/3)

- Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien que dans une portion confinée de celui-ci (à l'intérieur d'une fonction par exemple), on parle de **portée (ou visibilité) d'une variable**.

- **Une variable globale:** est déclarée au début du code, c'est-à-dire avant tout bloc de donnée, on pourra alors l'utiliser à partir de n'importe quel bloc d'instructions.

- **Une variable locale:** est déclarée à l'intérieur d'un bloc d'instructions (dans une fonction ou une boucle par exemple) aura une portée limitée à ce seul bloc d'instruction, c'est-à-dire qu'elle est inutilisable ailleurs.

Portée des variables (2/3)

- En C++, il existe un opérateur, appelé **opérateur de résolution de portée**, qui permet d'accéder à des variables globales plutôt qu'à des variables locales, lorsqu'il en existe portant le même nom.

- Cet opérateur se note **::**.

- Imaginons que l'on définisse plusieurs variables portant le même nom dans des blocs différents :

```
int i = 3;

void main(){
        int i = 12;

        ::i = i + ::i;
}
```

Portée des variables (3/3)

- **L'opérateur de résolution de portée** placé devant le nom de la variable indique qu'il s'agit de la variable globale.

- Ainsi, l'instruction ***::i = i + ::i*** , permet d'ajouter à la variable globale *i* (celle située en dehors de la fonction *main()*) la valeur de la variable locale *i*.

➔ **D'une manière générale il est tout de même préférable de donner des noms différents aux variables locales et globales...**

Définition de constantes (1/3)

- **Une constante** est une variable dont la valeur est inchangeable lors de l'exécution d'un programme.

- En langage C, les constantes sont définies grâce à la directive du préprocesseur **#define**, qui permet de remplacer toutes les occurrences du mot qui le suit par la valeur immédiatement derrière elle.

- Par exemple la directive suivante remplacera toutes les chaînes de caractères « _Pi » par la valeur 3.1415927 :

```
#define   _Pi   3.1415927
```

Définition de constantes (2/3)

- Toutefois, avec cette méthode les constantes ne sont pas typées, c'est pourquoi le C++ rajoute le mot réservé ***const***, qui permet de définir une constante typée et ne pourra pas être modifiée pendant toute la durée d'exécution du programme.

- Il est ainsi nécessaire d'initialiser la constante dès sa définition :

```
const  int  A  =  20;

const  float  Pi  =  3.14159
```

Définition de constantes (3/3)

- Exemple:

```
int x = 4;
const int y = 4;

 x += 1;     // x recevra la valeur 5
 y += 1;    // Cette opération est interdite par un compilateur C++ conforme
```

Les types entiers (1/2)

- Le langage C++ possède plusieurs types de base pour désigner un **entier**.

  - ✓ **int** : contient un entier de taille normale, positif ou négatif.
  - ✓ **short int** : contient un entier de petite taille, positif ou négatif.
  - ✓ **long int** : contient un entier de grande taille (32 bits), positif ou négatif.
  - ✓ **long long int** : contient un entier de plus grande taille (64 bits), positif ou négatif.
  - ✓ **unsigned int** : contient un entier de taille normale, positif ou nul.
  - ✓ **unsigned short int** : contient un entier de petite taille, positif ou nul.
  - ✓ **unsigned long int** : contient un entier de grande taille (32 bits), positif ou nul.
  - ✓ **unsigned long long int** : contient un entier de plus grande taille (64 bits), positif ou nul.

Les types entiers (2/2)

- Il est usuel de représenter un **int** sur 32 bits : il peut alors représenter n'importe quel entier entre -2^{31} et $2^{31}-1$.

- Le langage impose juste que la taille d'un **long int** doit être supérieure ou égale à celle d'un **int** et que la taille d'un **int** doit être supérieure ou égale à celle d'un **short int**.

- En C++ il existe plusieurs types d'entiers, dépendant du nombre d'octets sur lesquels ils sont codés ainsi que de leur format, c'est-à-dire s'ils sont **signés** (possédant le signe - ou +) ou non. Par défaut les données sont signées.

Les types réels

- Pour représenter un réel, il existe 3 types de base :
  - ✓ **float** (simple précision)
  - ✓ **double** (double précision)
  - ✓ **long double** (précision étendue)

Les caractères

- **Le type char:**
  - ✓ Il s'agit du type historique pour représenter un caractère.
  - ✓ Exemple :

```
Char c;
c= 'A';
```

- **Les types signed char et unsigned char:**

  - ✓ Lorsqu'on transfère un char dans un int, peut-on récupérer une valeur négative ?
    - ➔ La réponse est **oui** si on utilise le type **signed char** et **non** si on utilise le type **unsigned char**. Ces types peuvent être utile lorsqu'on manipule des caractères non ASCII.

  - ✓ Pour les données de type char, lorsque ni signed ni unsigned ne sont précisés, le choix entre les deux est fait par le compilateur.

Les booléens

- Le C++ utilise le type **bool** pour représenter une variable booléenne.
- Un bool ne peut prendre que 2 valeurs : true ou false.
- Exemple:

```
bool a;
int c;
 c = 89;
 a = (c > 87);
// a reçoit alors la valeur true.
```

L'opérateur sizeof

- L'opérateur **sizeof** permet de savoir le nombre d'octets qu'occupe en RAM une certaine variable.
- On peut écrire **sizeof(a)** pour savoir le nombre d'octets occupé par la variable a.
- On peut aussi écrire, **sizeof(int)** pour connaître le nombre d'octets occupés par une variable de type int.
- Cet opérateur est très utile lorsqu'on veut résoudre des problèmes de portabilité d'un programme.
- Plus précisément **sizeof(char)** vaut toujours **1**, par définition. Sur la plupart des architectures un char est codé par huit bits, soit un octet.

Définition d’un alias de type (1/2)

- Il est possible en C++ comme en C de définir un nouveau type de données grâce au mot clé **typedef**.

- Ceci permet dans certains cas de raccourcir le code, et dans tous les cas c'est l'occasion de donner un nom plus explicite à un type.

- Ceci favorise une meilleur lecture du code.

- La syntaxe de typedef est exactement la même que celle de la déclaration d'une variable, excepté que l'instruction commence par typedef et qu'aucune variable n'est réservée en mémoire, mais un alias du type est créé.

Définition d’un alias de type (2/2)

- Par convention, les noms typedef sont déclarés en utilisant un suffixe "**_t**".

➔ Cela aide à indiquer qu'ils sont des types, pas des variables, et aide également à prévenir les collisions des noms avec des variables de même nom.

```
typedef   double   distance_t;      // definir distance_t  comme un alias pour le type
                                       double
```

Les types de données en langage C++

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	Entier court non signé
Int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$-3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant double	8	$-1.7*10^{-308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$-3.4*10^{-4932}$ à $3.4*10^{4932}$
bool	Booléen	Même taille que le type *int*, parfois *1* sur quelques compilateurs	Prend deux valeurs : *'true'* et *'false'* mais une conversion implicite (valant 0 ou 1) est faite par le compilateur lorsque l'on affecte un entier (en réalité toute autre valeur que 0 est considérée comme égale à *True*).

Les entrées-sorties en C++

Rappel

- **Les entrées** sont les données que l'utilisateur fournit à un programme durant l'exécution, par l'intermédiaire du clavier ou d'un fichier.

- **Les sorties** sont les résultats que le programme fournit à l'utilisateur durant l'exécution, par l'intermédiaire de l'écran ou d'un fichier.

Classes de gestion des flux

- Les entrées et sorties sont gérées par deux classes définies dans le fichier d'en-tête **<iostream>** :

  ✓ **ostream** (Output stream) permet d'écrire des données vers la console, un fichier, ...
  Cette classe surdéfinit l'opérateur **<<**.

  ✓ **istream** (Input stream) permet de lire des données à partir de la console, d'un fichier, ...
  Cette classe surdéfinit l'opérateur **>>**.

Flux standards

- Trois instances de ces classes représentent les flux standards :

  - ✓ **cout** écrit vers la sortie standard,
  - ✓ **cerr** écrit vers la sortie d'erreur,
  - ✓ **cin** lit à partir de l'entrée standard.

- Ces trois objets sont définis dans l'espace de nom **std**.

Le flux sortant

- Pour afficher à l'écran la valeur d'une variable x :

```
cout << x << endl;
```

➔ Le **endl** signifie que le programme devra aller à la ligne après la valeur de x.

- Pour ajouter des commentaires, par exemple : Valeur de x :

```
cout << "Valeur de x : " << x << endl;
```

- Pour afficher les valeurs de plusieurs variables x, y et z :

```
cout << x << " " << y << " " << z << endl;
```

➔ Il faut au moins un commentaire blanc pour séparer les valeurs des variables.

Le flux entrant (1/2)

- Pour attribuer une valeur à une variable **x** à partir du clavier :

```
cin >> x;
```

- Lors de l'exécution, la valeur entrée au clavier doit être validée par un Entrée. Il faut faire précéder cette instruction d'une demande telle que :

```
cout << "Valeur de x ? ";
```

- Pour lire plusieurs valeurs à la fois :

```
cout << "Valeur de x y z ? ";
cin >> x >> y >> z;
```

Le flux entrant (2/2)

- **Problème:** si on essaye de lire une phrase complète contenant des espaces. En utilisant l'opérateur **>>** sur un flux entrant, la lecture s'arrête au premier "blanc".

  ➔ Pour lire une phrase, il faut alors passer par la fonction **getline()**.

- **Exemple:**

```
cout << "Entrez une phrase: ";

string phrase;

getline(cin,phrase);     //Copie toute la ligne de ce qui se trouve dans le flux dans la
                         //chaîne phrase.
```

Exemple

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

 int main() {
        int n;
        cout << "Entrez un nombre positif : ";
        cin >> n;
        if (n<0)
            cerr << "Erreur: Le nombre " << n << " n'est pas positif " << endl;
       else
           cout << "Vous avez entré " << n << endl;
        return 0;
}
```

Autres types de flux (1/2)

- En C++, il existe plusieurs manières pour un programme d'interagir avec les éléments extérieurs que sont: la console, les fichiers, les périphériques, etc.

- Un des moyens particulièrement puissant d'interaction est le **flux** ou **flot**.

- **Il existe des flux dans deux directions:** du programme vers l'extérieur et évidemment dans le sens contraire, ce qui permet ainsi d'influencer le déroulement d'un programme.

- La bibliothèque standard du C++ propose 3 types différents de flux ayant chacun 2 directions. Ces flux sont des instances de classes dont les noms sont présentés dans le tableau suivant :

Autres types de flux (2/2)

Type de flux	Flux entrant	Flux sortant	Fichier à inclure
Entrée/Sortie standard (console)	std::istream	std::ostream	**iostream**
Flux sur les fichiers	std::ifstream	std::ofstream	**fstream**
Flux sur les chaînes de caractères	std::istringstream	std::ostringstream	**sstream**

Le dernier type de flux n'est pas fait pour interagir avec le monde extérieur mais plutôt pour utiliser la puissance des flux pour gérer plus facilement les chaînes de caractères.

Flux de fichier (1/6)

❑ La classe **ifstream** permet de lire à partir d'un fichier. Le constructeur a la syntaxe suivante :

```
ifstream ( const char* filename , openmode mode = in )
```

- Le paramètre **mode** peut être une combinaison des valeurs suivantes :

  ✓ **app (append):** placer le curseur à la fin du fichier avant écriture.

  ✓ **ate (at end):** placer le curseur à la fin du fichier.

Flux de fichier (2/6)

- La différence entre **app** et **ate**:

  ✓ **app** permet d'ajouter à la suite: lorsqu'on ouvre le fichier en écriture, on se trouve à la fin pour écrire des données à la suite du fichier sans effacer le contenu, s'il y en a un.

  ➔ Avec ce mode d'ouverture, à chaque écriture, on est placé à la fin du fichier, même si on se déplace dans celui-ci avant.

  ✓ **ate**: ouvre le fichier en écriture et positionne le curseur à la fin de celui-ci.

  ➔La différence avec app est que si on se repositionne dans le fichier, l'écriture ne se fera pas forcément à la fin du fichier, contrairement à app.

Flux de fichier (3/6)

- ✓ **Binary:** ouvrir en mode binaire plutôt que texte.
- ✓ **in:** autoriser la lecture.
- ✓ **out:** autoriser l'écriture.
- ✓ **trunc (truncate):** tronquer le fichier à une taille nulle.

- **Exemple 1:** lire un entier depuis un fichier

```
ifstream  fichier ("test.txt");
int a;
fichier >> a;        // lire un entier
cout  <<  "A = "  <<  a;
fichier.close();
```

Flux de fichier (4/6)

- **Exemple 2:** afficher tous les caractères d'un fichier

```
ifstream fichier("test.txt");

while (fichier.good())
    cout << (char) fichier.get();

fichier.close();
```

- good(): vérifier si l'état du flux est bon

Flux de fichier (5/6)

- La classe **ofstream** permet d'écrire vers un fichier. Son constructeur a une syntaxe similaire:

```
ofstream (const char* filename , openmode mode=out|trunc )
```

- Exemple:

```
ofstream fichier ("test.txt");

fichier << setw(10) << a << endl;   //Ajout de 10 caractères espaces

fichier.close();
```

Flux de fichier (6/6)

❑ La classe **fstream** dérive de la classe **iostream** permettant à la fois la lecture et l'écriture.

- **iostream** dérive donc à la fois de la classe ostream et de la classe istream. Son constructeur a la syntaxe suivante :

```
fstream (const char* filename, openmode mode=in|out )
```

Flux de chaîne de caractères (1/6)

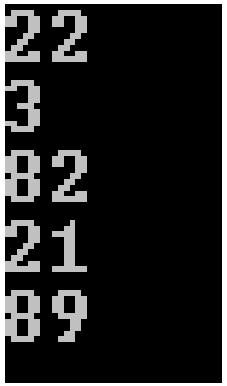
- La classe **istringstream** dérivée de **istream** permet de lire à partir d'une chaîne de caractères et possède deux constructeurs:

```
istringstream ( openmode  mode = in );

istringstream ( const  string & str,  openmode  mode = in );
```

Flux de chaîne de caractères (2/6)

- Exemple:

```
int i, val;

string  chaine;

chaine=  "22 3 82 21 89";

istringstream iss (chaine, istringstream::in);

for (i = 0; i < 5; i++) {

        iss >> val;
        cout << val << endl;

}
```



Flux de chaîne de caractères (3/6)

❑ La classe **ostringstream** dérivée de **ostream** permet d'écrire pour créer une chaîne de caractères et possède également deux constructeurs :

```
ostringstream ( openmode mode = out );

ostringstream ( const string & str, openmode mode = out );
```

- Le second constructeur permet de spécifier le début de la chaîne de caractères produite.

Flux de chaîne de caractères (4/6)

- Exemple:

```
ostringstream oss (ostringstream::out);

int a = 20;

oss << "La valeur de a=" << a << "\n";

cout << oss.str();
```

La méthode **str()** retourne la chaîne de caractères produite.

```
La valeur de a=20
```

Flux de chaîne de caractères (5/6)

- La classe **stringstream** dérivée de **iostream** permet d'écrire et lire, et possède deux constructeurs :

```
stringstream ( openmode mode = in | out );

stringstream ( const string & str , openmode mode = in | out );
```

Flux de chaîne de caractères (6/6)

- Exemple:

```
int i, val;

stringstream ss (stringstream::in | stringstream::out);

 // écriture
 ss << "22 3 14 82 21 89 ";

// lecture
for (int i = 0; i < 6; i++) {

      ss >> val;
      cout << val << endl;

}
```

Les manipulateurs en C++

Les manipulateurs basiques

❑ **Syntaxe:**

- Un manipulateur de flux est une instruction permettant de modifier le comportement du flux dans certains cas.

- Il existe deux syntaxes équivalentes pour utiliser un manipulateur.
  - ✓ La première manière correspond à un appel d'une fonction :

```
nom_du_manipulateur(nom_du_flux);
```

  - ✓ La deuxième s'appuie sur la surcharge de l'opérateur << pour les flux :

```
nom_du_flux << nom_du_manipulateur;
```

Le manipulateur boolalpha

- Le manipulateur **boolalpha** permet d'afficher **true** et **false**, plutôt que **1** et **0**.

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

int main()
{
   cout << true << " et " << false << endl;  //Affichage normal
   cout << boolalpha;                        //On applique le manipulateur
   cout << true << " et " << false << endl;  //Affichage "modifié"

   return 0;
}
```

```
1 et 0

true et false
```

- **noboolalpha** est le manipulateur opposé.

Les manipulateurs sur les nombres entiers (1/2)

- Un même nombre peut être représenter selon plusieurs manières tout dépend de la base choisie (base décimal (10), octal (8), hexadécimal (16)).

- Pour cela, il existe 3 manipulateurs:

Nom	Fonction
dec	Affiche les nombres en base 10. (Valeur par défaut)
oct	Affiche les nombres en base 8.
hex	Affiche les nombres en base 16.

Les manipulateurs sur les nombres entiers (2/2)

- **dec** : Ce manipulateur indique que les prochains entiers sont à lire ou écrire en base décimale.
- **hex** : Ce manipulateur indique que les prochains entiers sont à lire ou écrire en base hexadécimale.
- **oct** : Ce manipulateur indique que les prochains entiers sont à lire ou écrire en base octale.

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

int main()
{
   cout << 45 << endl;              //Affichage "normal" en base 10
   cout << oct << 45 << endl;       //Affichage  en base 8
   cout << hex << 45 << endl;       //Affichage  en base 16
   cout << dec << 45 << endl;       //Retour à la base 10

   return 0;
}
```

```
45

55

2D

45
```

Les manipulateurs sur les nombres à virgule (1/2)

- Un nombre à virgule peut s'afficher par plusieurs façons: en affichant tous les chiffres ou en utilisant la notation scientifique (**scientific**, **fixed**).

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

int main()
{
   double nombre =  452.214857;

   cout << nombre << endl;                  //Affichage normal
   cout << fixed << nombre << endl;         //Affichage de tous les chiffres
   cout << scientific << nombre << endl;  //Affichage scientifique

   return 0;
}
```

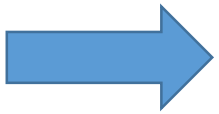
```
452.215

452.214857

4.522149e+02
```

Les manipulateurs sur les nombres à virgule (2/2)

- **showpoint**: afficher le **.** séparant la partie entière et décimale même si le nombre ne possède pas de partie décimale.
- **noshowpoint** est son opposé.
- **showpos**: forcer l'affichage du signe **+** pour les nombres positifs.
- **noshowpos**: son opposé
- **setprecision(digits):** Ce manipulateur spécifie que les prochains nombres à virgule flottante doivent être écrits avec la précision donnée. La précision donne le nombre maximum de chiffres à écrire (avant et après la virgule).

- Les créateurs de la bibliothèque standard du C++ ont mis 6 manipulateurs dans un autre fichier, **iomanip**.
- Tous les manipulateurs dont le nom commence par **set...** sont définis dans ce fichier.

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

int main()
{
   double pi = 3.141592653 ;

   cout << pi << endl;
   cout << setprecision(0) << pi << endl;
   cout << setprecision(10) << pi << endl;

   return 0;
}
```



```
3.14159

3

3.141592653
```

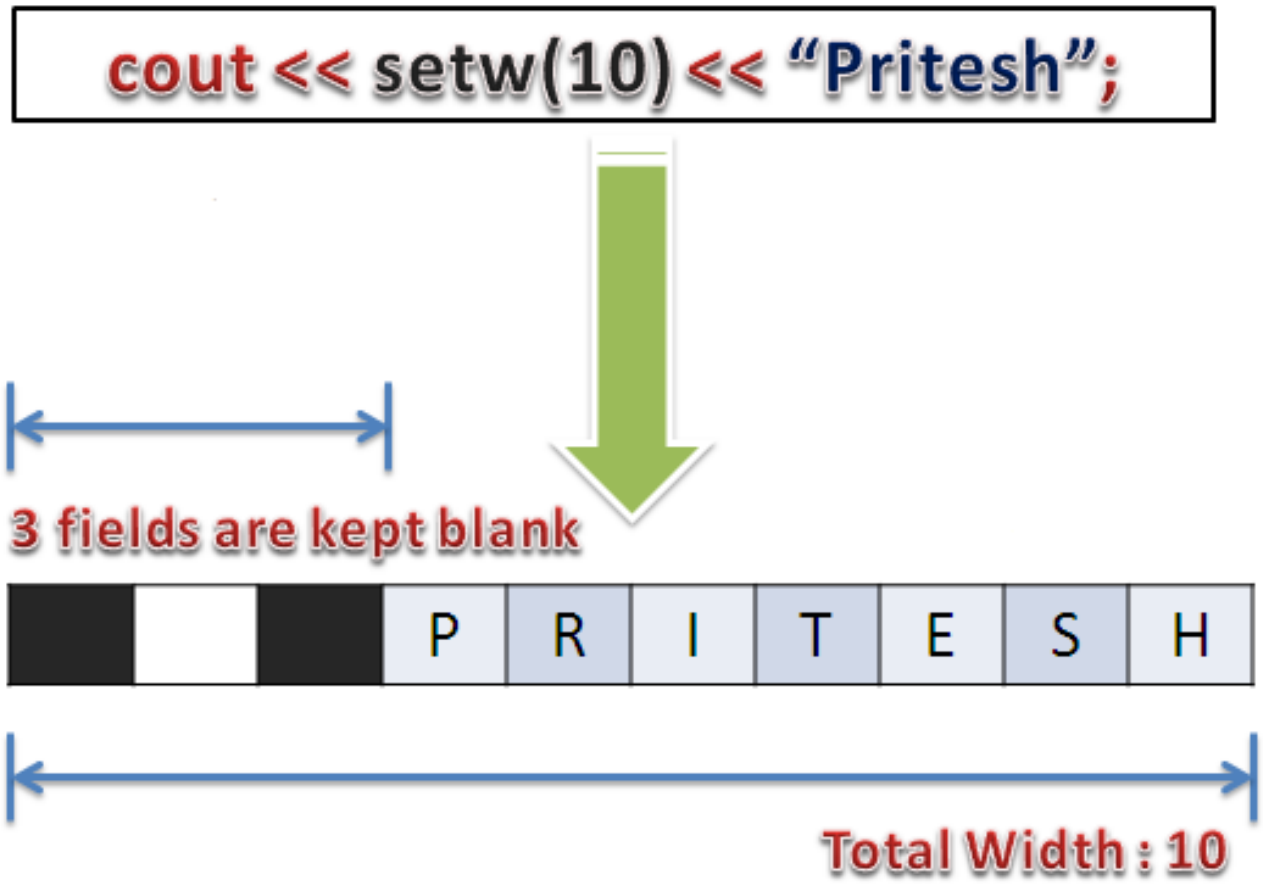
Quelques manipulateurs plus avancés (1/5)

- **setw(width)**: Ce manipulateur indique que les prochaines données doivent être écrites sur le nombre de caractères indiqué, en ajoutant des caractères espaces avant.

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

int main()
{
cout << "Alignement normal" << endl;
cout << setw(10) << abcd<<  endl;    //On aligne tout dans une "case" de 10 caractères

}
```

```
Alignement normal
      abcd
```

Quelques manipulateurs plus avancés (2/5)



Quelques manipulateurs plus avancés (3/5)

- **left**: aligner la sortie à gauche lors de l'utilisation de setw.
- **right**: aligner la sortie à droite lors de l'utilisation de setw (comportement par défaut).

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

int main()
{
cout << "Mon premier tableau" << endl << endl;
cout << setw(10) << left << "Nom: " << "|" << setw(8) << right << "dec" << endl;
cout << setw(10) << left << "Base: " << "|" << setw(8) << right << "base10" << endl;
}
```

```
Mon premier tableau

Nom:       |     dec
Base:      |  base10
```

Quelques manipulateurs plus avancés (4/5)

- **setfill(char)**: changer le caractère de remplissage.

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

int main()
{
cout << "Un joli tableau" << endl << endl;
cout << setfill('.');
cout << setw(10) << left << “Nom: " << "|" << setw(8) << right << “dec" << endl;
cout << setw(10) << left << “Base: " << "|" << setw(8) << right << “base10" << endl;
}
```

```
Un joli tableau

Nom: .....|.....dec
Base: ....|..base10
```

Quelques manipulateurs plus avancés (5/5)

- **endl:** Ce manipulateur écrit un retour à la ligne dans le flux, quel qu'il soit (\r\n pour Windows, \n pour Unix/Linux, \r pour Mac, ...).

```
Cout  <<  "Une première ligne"    << endl  <<  "Une deuxième ligne"  << endl;
```

Les structures de contrôle en C++

Introduction

- Les conditions permettent de tester des variables en utilisant des symboles.

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

La condition de type if (1/2)

- Cette structure de contrôle permet d'exécuter une instruction ou une suite d'instructions seulement si une condition est vraie.

```
if (condition) instruction
```

- **La condition:** expression booléenne de la condition d'exécution.

- **L'instruction:** une instruction ou un bloc d'instructions exécuté si la condition est vraie.

- On évalue la condition :
  - ✓ si elle est vraie on exécute l'instruction et on passe à l 'instruction suivante.
  - ✓ si elle est fausse on passe directement à l 'instruction suivante.

- L'instruction peut être remplacée par une suite d 'instructions entre accolades.

La condition de type if (2/2)

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

int main() {
	int x = 102;

	if (x> 100)
	cout << " x est plus grand que 100" << endl;

	cout << " Fin du programme !" << endl;

	return 0;
 }
```

```
x est plus grand que 100
 Fin du programme !
```

La condition de type if…else (1/2)

- Cette structure de contrôle permet d'exécuter une instruction si une condition est vraie, ou une autre instruction si elle est fausse.

```
if  (condition)  instruction1

else   instruction2
```

- **La condition:** expression booléenne de la condition d'exécution.

- **L'instruction 1:** une instruction ou un bloc d'instructions exécuté si la condition est vraie.

- **L'instruction 2:** une instruction ou un bloc d'instructions exécuté si la condition est fausse.

La condition de type if…else (2/2)

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

int main() {
	int x = 88;

	if (x> 100)
	cout << " x est plus grand que 100" << endl;
	else
	cout << " x est inférieur à 100" << endl;

	cout << " Fin du programme !" << endl;

	return 0;
 }
```

```
x est inférieur à 100
Fin du programme !
```

La condition de type switch (1/2)

- L'instruction switch permet de tester plusieurs valeurs pour une expression.

```
switch (expression)
{
case constante1:
      instruction1_1
      instruction1_2...

case constante2:
      instruction2_1
      instruction2_2...
 ...

default:
      instruction_1
      instruction_2...
}
```

- **L'expression:** expression de type scalaire (entier, caractère, énumération, booléen).
- **case constante1: instruction1_...:** une série d'instructions exécutés si expression vaut constante1.
- **case constante2: instruction2_...:** une série d'instructions exécutés si expression vaut constante2.
- **default: instruction_...:** une série d'instructions exécutés quand aucun des cas précédents ne correspond.

La condition de type switch (2/2)

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

int main() {

      int x;
      cout << "Tapez la valeur de x : ";
      cin >> x;

       switch(x) {

         case 1 :
            cout << “x vaut 1" << endl;
            break;

         case 2 :
            cout << “x vaut 2" << endl;
            break;

         default :
            cout << “x ne vaut ni 1, ni 2," << endl;
            break;
       }

       return 0;

}
```

La boucle for (1/2)

- La boucle **for** est une structure de contrôle qui permet de répéter un certain nombre de fois une partie d'un programme.

```
for (instruction_init ; condition ; instruction_suivante)

    instruction(s)
```

- **L'instruction_init:** une instruction d'initialisation de la boucle (initialiser un compteur à 0 par exemple).

- **La condition:** expression booléenne de la condition de répétition de la boucle.

- **L'instruction_suivante:** une instruction pour passer à l'itération suivante (passer à l'élément suivant, ...)

- **L'instruction(s):** un bloc d'instruction répété à chaque itération de la boucle.

La boucle for (2/2)

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

int main() {

        int i;

        for (i=0 ; i<10 ; i++)
          cout << "Bienvenue" << endl;

        return 0;

      }
```

La boucle while (1/2)

```
while  (condition)

{  instruction(s);

}
```

- On teste la condition :
  - ✓ si elle est vraie, on exécute l'instruction et on recommence.
  - ✓ si elle est fausse, la boucle est terminée, on passe à l'instruction suivante.

- Tout ce qui est entre accolades sera répété tant que la condition est vérifiée.

La boucle while (2/2)

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

int main() {

      int x =  0;

      while (x < 5)
      {
         cout << "La valeur de x est : " << x << endl;
         x++;
       }

       cout << "La valeur finale de x est : " << x << endl;
       return 0;

      }
```

```
La valeur de x est : 0

La valeur de x est : 1

La valeur de x est : 2

La valeur de x est : 3

La valeur de x est : 4

La valeur finale de x est : 5
```

La boucle do…while (1/2)

```
do
{
 ...instructions...

} while ( condition );
```

1. exécution des instructions ;
2. évaluation de la condition ;
3. si elle est vraie, on recommence au 1 ;
4. si elle est fausse, la boucle est terminée, on passe à l ’instruction suivante.

La boucle do…while (2/2)

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

int main()  {

      int x = 0;

      do {

       cout << "La valeur de x vaut : " << x << endl;
       x = x + 1;

       } while (x < 5);

      cout << "La valeur finale de x est " << x << endl;
      return 0;

      }
```

```
La valeur de x vaut : 0

La valeur de x vaut : 1

La valeur de x vaut : 2

La valeur de x vaut : 3

La valeur de x vaut : 4

La valeur finale de x est : 5
```

Le break

- L'instruction **break** sert à "casser" ou interrompre une boucle (for, while et do), ou un switch.

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

int main()  {
      int x;
      for (int x = 0; x < 10; x++)
      {
          cout << "La variable x vaut: " << x << endl;
          if (x == 2)
              break;
      }

      return 0;
      }
```

```
La variable x vaut: 0

La variable x vaut: 1

La variable x vaut: 2
```

Le continue

- L'instruction **continue** sert à "continuer" une boucle (for, while et do) avec la prochaine itération.

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

int main()  {
      for (int x = 0; x < 5; x++)
       {
            if (x == 2) continue;

            cout << "La variable x vaut: " << x << endl;

         }
      return 0;
}
```

```
La variable x vaut: 0

La variable x vaut: 1

La variable x vaut: 3

La variable x vaut: 4
```

Les outils de programmation nécessaires en C++

Introduction

- Pour programmer il faut:

  ✓ **Un éditeur de texte** pour écrire le code source du programme en C++ (comme le Bloc-Notes sous Windows ou **vi** sous Linux).

  ➔L'idéal, c'est d'avoir un éditeur de texte intelligent qui colore tout seul le code, ce qui vous permet de vous y repérer bien plus facilement. C'est pourquoi aucun programmeur n'utilise le Bloc-Notes.

  ✓ **Un compilateur** pour transformer (compiler) votre code source en binaire.

  ✓ **Un débugger** (« Débogueur » ou « Débugueur » en français) pour vous aider à traquer les erreurs dans votre programme.

- Le langage C++ fait partie des langages compilés : le fichier exécutable est produit à partir de **fichiers sources** par **un compilateur**.

Les fichiers entêtes

- Les fichiers "**entêtes**" ("**headers**" en anglais), traditionnellement d'extension **.h** ou **.hpp** contiennent généralement les prototypes de différentes fonctions, structures et classes.

- Ces fichiers sont inclus dans les fichiers sources à l'aide de cette directive:

```
#include " nom_header.h "
```

Fichiers sources

- Les fichiers "**sources**" d'extension **.cpp** (parfois **.c** ou **.cc**) contiennent la définition (l'implémentation) des différentes fonctions et méthodes définies dans les fichiers d'en-tête.

- La compilation des fichiers **.cpp** produit dans un premier temps des fichiers objets (extension **.obj** ou **.o** en général).

- Les fichiers **.cpp** utilisent les fichiers d'en-tête.

- Ils les appellent en utilisant la syntaxe **#include "nomdefichier.h"**

Programmation C++ : Compilation

- **La compilation:** Série d'étapes de **transformation** du **code source** en du **code machine** exécutable sur un processeur cible.

- Le **langage C++** fait partie des **langages compilés**: le fichier exécutable est produit à partir de **fichiers sources** par **un compilateur**.

Fonctionnement du compilateur (1/2)

- Le compilateur réalise les trois grandes étapes successives suivantes :

1. **Précompilation**: Le code source original est transformé en code source brut.

   Les commentaires sont enlevés et les directives de compilation commençant par **#** sont d'abord traités pour obtenir le code source brut.

2. **Compilation en fichier objet**: vérification syntaxique et concordance des types.

   Si aucune erreur décelée, les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante.

   Généralement, ces fichiers portent l'extension **.obj** ou **.o** .

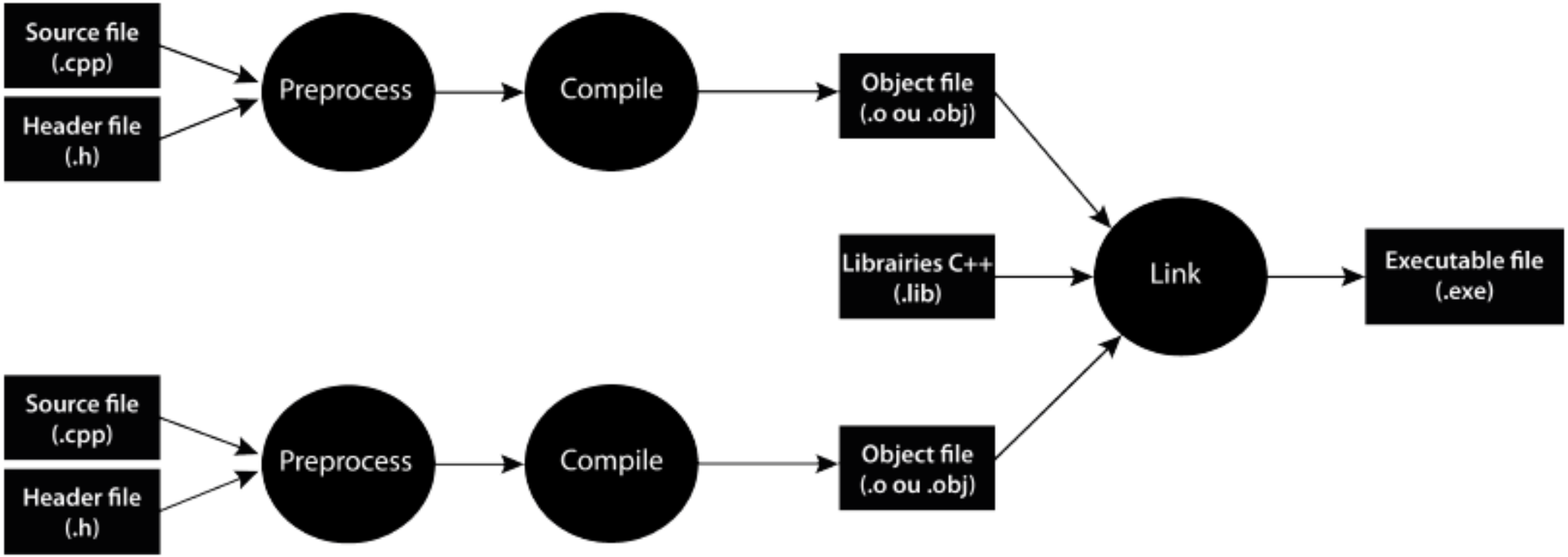
Fonctionnement du compilateur (2/2)

3. **Édition des liens** : la compilation a permis de pointer où se trouvait la définition de fonction externe (ex : #include <iostream>) pour satisfaire aux étapes de vérification.

Cependant les instructions en langage machine de ces modules externes n'ont pas été ajoutées, car elles sont déjà définies dans d'autres fichiers (les **.lib**) comprenant leurs instructions machine.

L'étape d'édition des liens rassemble toutes les instructions machine du projet et modules externes en un fichier exécutable (**.exe**).

Vue globale



Chaine de traitement du compilateur

Les IDEs

- La liste de quelques IDEs parmi les plus connus et qui sont disponibles gratuitement:

  ✓ **Code::Blocks**: Il est gratuit et disponible, fonctionne sous Windows, Mac et Linux.

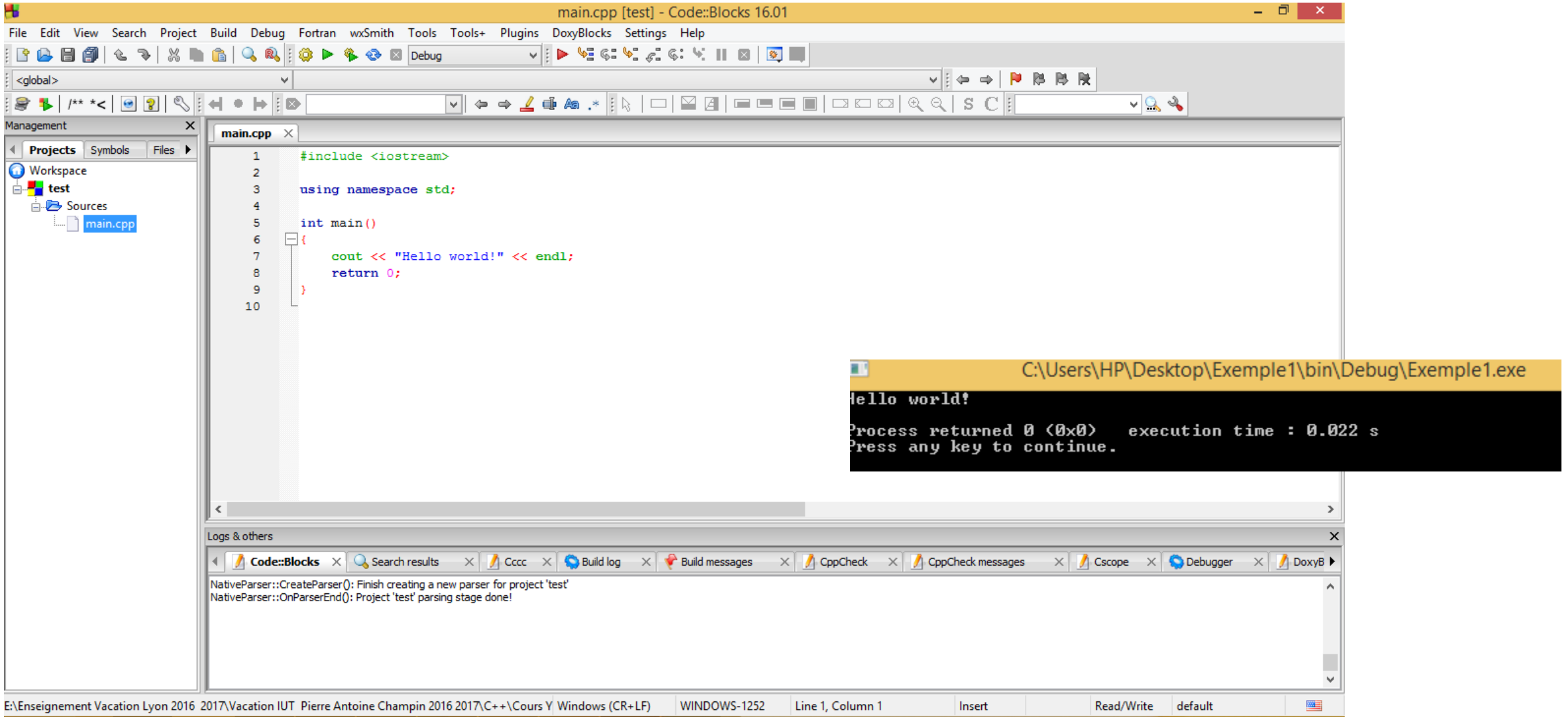
  ✓ **Visual C++**: IDE de Microsoft, le plus célèbre IDE sous Windows.

  - Il existe une version gratuite intitulée **Visual C++ Express**. Il est très complet et possède un puissant module de correction des erreurs (débuggage).

  - Il fonctionne uniquement sous Windows.

  ✓ **Xcode**: Il fonctionne sous Mac OS X uniquement.

Exemple avec Code::Blocks



TP 1