Algorithmique
Cours 3

IUT Informatique de Lens, 1ère Année
Université d'Artois

Frédéric Koriche
koriche@cril.fr
2011 - Semestre 1

Sommaire

L'objectif de ce cours est d'étudier le langage C++ pour les types de données simples et les principales instructions.

Sommaire

L'objectif de ce cours est d'étudier le langage C++ pour les types de données simples et les principales instructions.

Compilateur

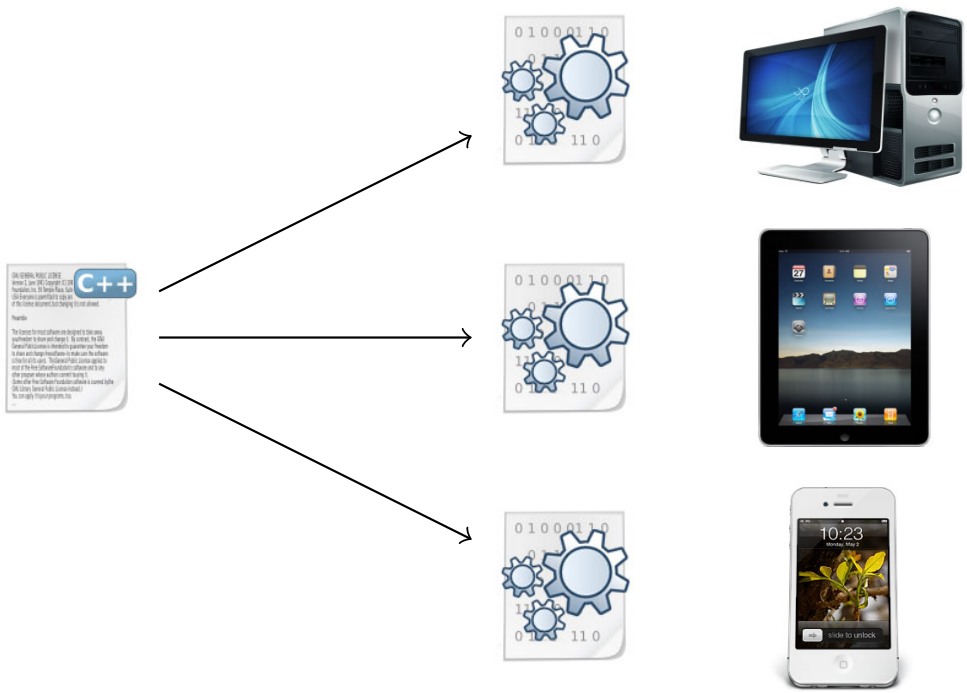
Un compilateur est un programme informatique qui transforme

- un **code source** écrit dans un langage de programmation compréhensible par l'humain
- en un **code assembleur** ou **code machine** compréhensible seulement par la machine

Compilateur

Un compilateur est un programme informatique qui transforme

- un **code source** écrit dans un langage de programmation compréhensible par l'humain
- en un **code assembleur** ou **code machine** compréhensible seulement par la machine

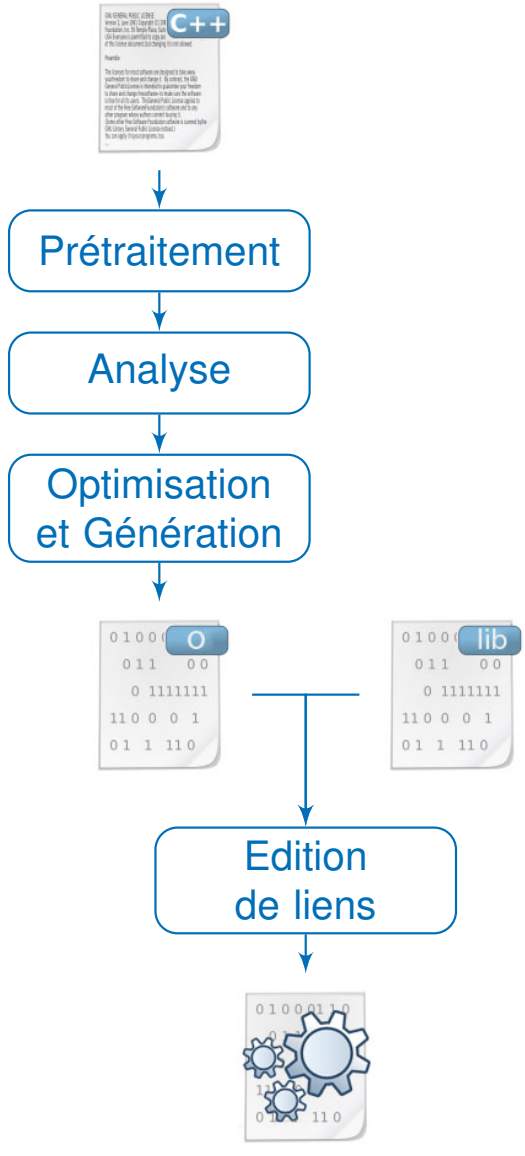


Un algorithme est (théoriquement) portable sur toute machine. Le compilateur permet de traduire le code source de l'algorithme en un code assembleur propre à chaque machine.

Compilateur

Les phases d'un compilateur C ou C++ comprennent

- le **prétraitement** qui inclut les fichiers en-tête (.h) au code source (.cpp), prend en charge les macros et gère la compilation conditionnelle.
- l'**analyse** (lexicale, syntaxique et sémantique) qui transforme le code source en code intermédiaire (avec une table des symboles)
- l'**optimisation** et la **génération** qui transforment le code intermédiaire en code objet (.o)
- l'**édition de liens** qui intègre au code objet les librairies (.lib) afin de produire le code assembleur (.bin ou .exe) final.



Données (rappel)

Toute donnée est spécifiée par :

- son **nom** : désigne la donnée dans le programme.
- son **type** : décrit le domaine de valeurs que peut prendre la donnée.
- sa **nature** : variable (la valeur peut changer) ou constante (la valeur est fixe).

Données (rappel)

Toute donnée est spécifiée par :

- son **nom** : désigne la donnée dans le programme.
- son **type** : décrit le domaine de valeurs que peut prendre la donnée.
- sa **nature** : variable (la valeur peut changer) ou constante (la valeur est fixe).

Codage des données

L'informatique utilise plusieurs codages de données, dont les principaux sont :

- **Décimal :** en mathématiques, les données sont généralement codées en base 10.
- **Binaire :** dans une mémoire électronique, les données sont codées en base 2.
- **Hexadécimal :** codage en base 16 (2^4), utilisé pour interpréter les données binaires.

Codage décimal

Tout entier positif x est codé par un tableau $\boldsymbol{x} =< x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{10} x + 1 \rfloor$ cases (ou chiffres),
- chacun ayant une valeur de 0 à 9,
- et tel que :

$$x = \sum_{i=0}^{n} 10^i x_i$$

Codage décimal

Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{10} x + 1 \rfloor$ cases (ou chiffres),
- chacun ayant une valeur de 0 à 9,
- et tel que :

$$x = \sum_{i=0}^{n} 10^i x_i$$

Exemple

Considérons le nombre entier x “un million cent mille neuf cent vingt quatre”. En base 10 il est codé par le tableau :

1	1	0	0	9	2	4
10^6	10^5	10^4	10^3	10^2	10^1	10^0

On a bien $1,100,924 = 10^6 \cdot 1 + 10^5 \cdot 1 + 10^4 \cdot 0 + 10^3 \cdot 0 + 10^2 \cdot 9 + 10^1 \cdot 2 + 10^0 \cdot 4$

Codage binaire

Tout entier positif x est codé par un tableau $\boldsymbol{x} =< x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_2 x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant la valeur 0 ou 1,
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 2^i x_i$$

Codage binaire

Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_2 x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant la valeur 0 ou 1,
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 2^i x_i$$

Exemple : Conversion décimale → binaire

Considérons le nombre entier x "soixante quinze". En base 2 il est codé par le tableau :

1	0	0	1	0	1	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

On a bien : $75 = 64 \cdot 1 + 32 \cdot 0 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 \cdot 1$

Codage binaire

Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_2 x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant la valeur 0 ou 1,
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 2^i x_i$$

Exemple : Conversion binaire → décimale

Considérons le nombre binaire représenté par le tableau :

1	1	0	0	0	1	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

Sa conversion décimale est $64 \cdot 1 + 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 0 + 4 \cdot 0 + 2 \cdot 1 + 1 \cdot 1 = 99$

Codage hexadécimal

Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{16} x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant une valeur dans $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 16^i x_i$$

Codage hexadécimal

Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{16} x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant une valeur dans $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 16^i x_i$$

Exemple : Conversion binaire → hexadécimale

Chaque paquet de 4 bits est regroupé et codé par un chiffre hexadécimal.

0	1	0	0	1	0	1	1

Codage hexadécimal

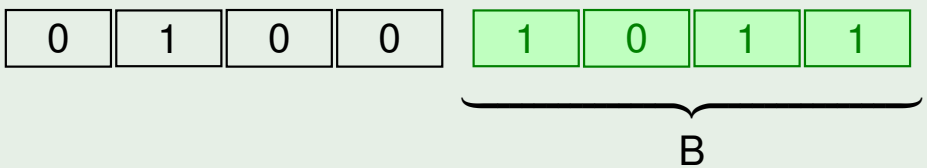
Tout entier positif x est codé par un tableau $\boldsymbol{x} =< x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{16} x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant une valeur dans $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 16^i x_i$$

Exemple : Conversion binaire → hexadécimale

Chaque paquet de 4 bits est regroupé et codé par un chiffre hexadécimal.



Codage hexadécimal

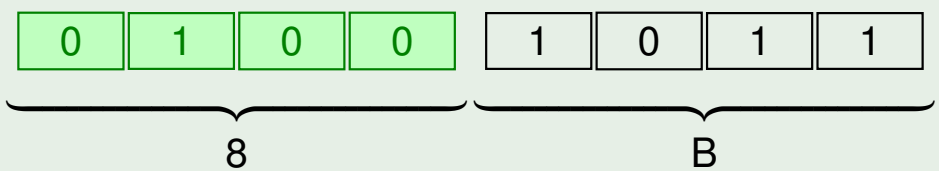
Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{16} x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant une valeur dans $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 16^i x_i$$

Exemple : Conversion binaire → hexadécimale

Chaque paquet de 4 bits est regroupé et codé par un chiffre hexadécimal.



Codage hexadécimal

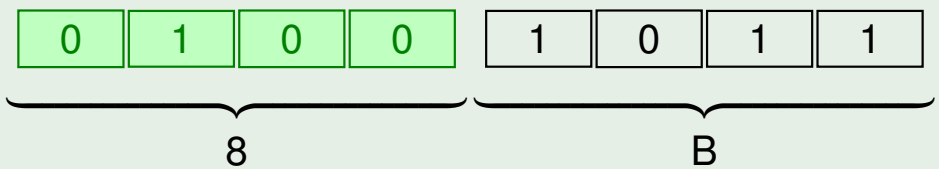
Tout entier positif x est codé par un tableau $\boldsymbol{x} = < x_1, \cdots, x_n >$

- comprenant $n = \lfloor \log_{16} x + 1 \rfloor$ cases (ou chiffres),
- chaque case ayant une valeur dans $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- et vérifiant l'égalité :

$$x = \sum_{i=0}^{n} 16^i x_i$$

Exemple : Conversion binaire → hexadécimale

Chaque paquet de 4 bits est regroupé et codé par un chiffre hexadécimal.



Le nombre décimal 75 correspond bien au nombre hexadécimal $8B$.

Types de données

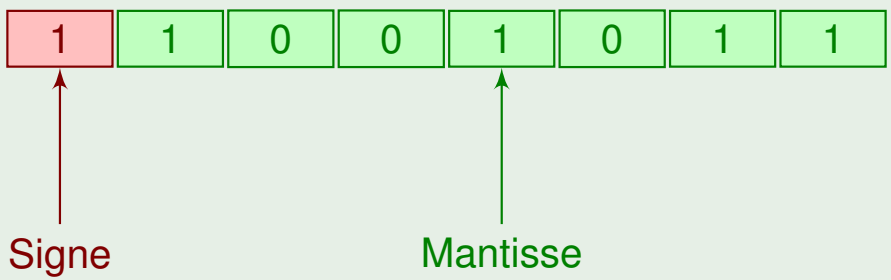
Type	C++	Longueur	Valeurs
Booléen	`bool`	1 octet	Valeurs 0 ou 1
Caractère	`char`	1 octet	Symboles ascii (0 à 255)
Entier Entier positif	`int` `unsigned int`	4 octets 4 octets	$[-2147483648, 2147483647]$ $[0, 4294967295]$
Réel	`float` `double`	4 octets 8 octets	$\pm 3.410^{\pm 38}$ (7 décimales) $\pm 1.710^{\pm 308}$ (15 décimales)

Types de données

Type	C++	Longueur	Valeurs
Booléen	`bool`	1 octet	Valeurs 0 ou 1
Caractère	`char`	1 octet	Symboles ascii (0 à 255)
Entier Entier positif	`int` `unsigned int`	4 octets 4 octets	$[-2147483648, 2147483647]$ $[0, 4294967295]$
Réel	`float` `double`	4 octets 8 octets	$\pm 3.410^{\pm 38}$ (7 décimales) $\pm 1.710^{\pm 308}$ (15 décimales)

Entiers relatifs

Le codage d'un entier relatif nécessite un bit d'information pour le signe. Par exemple, le codage binaire du nombre -75 est :

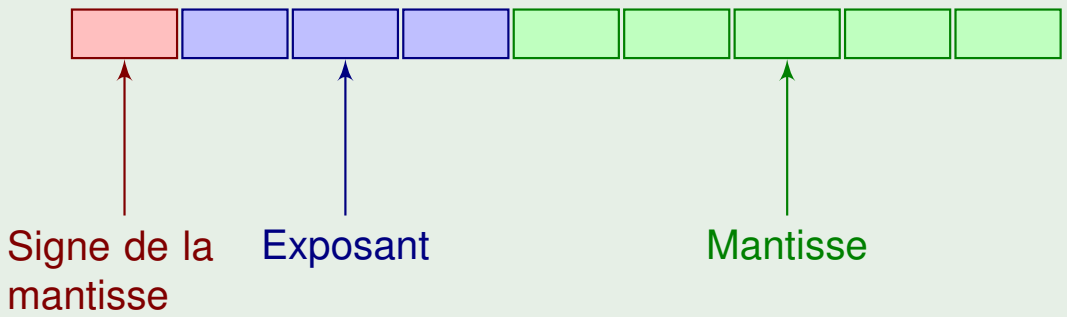


Types de données

Type	C++	Longueur	Valeurs
Booléen	`bool`	1 octet	Valeurs 0 ou 1
Caractère	`char`	1 octet	Symboles ascii (0 à 255)
Entier Entier positif	`int` `unsigned int`	4 octets 4 octets	$[-2147483648, 2147483647]$ $[0, 4294967295]$
Réel	`float` `double`	4 octets 8 octets	$\pm 3.410^{\pm 38}$ (7 décimales) $\pm 1.710^{\pm 308}$ (15 décimales)

Réels

Le codage d'un réel nécessite de représenter son exposant, sa mantisse et leur signe respectif.



Opérateurs arithmétiques (entiers)

Toutes les entrées sont des **entiers**
La sortie est un **entier**

Nom	C++
addition	+
soustraction	-
multiplication	*
division entière	/
reste	%
inversion de signe	-

Opérateurs arithmétiques (entiers)

Toutes les entrées sont des **entiers**
La sortie est un **entier**

Nom	C++
addition	+
soustraction	-
multiplication	*
division entière	/
reste	%
inversion de signe	-

Opérateurs arithmétiques (réels)

Au moins une entrée est un **réel**
La sortie est un **réel**

Nom	C++
addition	+
soustraction	-
multiplication	*
division réelle	/
inversion de signe	-

Opérateurs arithmétiques (entiers)

Toutes les entrées sont des **entiers**
La sortie est un **entier**

Nom	C++
addition	+
soustraction	-
multiplication	*
division entière	/
reste	%
inversion de signe	-

Opérateurs arithmétiques (réels)

Au moins une entrée est un **réel**
La sortie est un **réel**

Nom	C++
addition	+
soustraction	-
multiplication	*
division réelle	/
inversion de signe	-

Opérateurs de comparaison

Les deux entrées sont des **entiers**, **caractères** ou **réels**
La sortie est un **booléen**

Nom	C++
est égal à	==
est différent de	!=
est plus petit que	<
est plus grand que	>
est plus petit ou égal à	<=
est plus grand ou égal à	>=

Opérateurs arithmétiques (entiers)

Toutes les entrées sont des **entiers**
La sortie est un **entier**

Nom	C++
addition	+
soustraction	-
multiplication	*
division entière	/
reste	%
inversion de signe	-

Opérateurs arithmétiques (réels)

Au moins une entrée est un **réel**
La sortie est un **réel**

Nom	C++
addition	+
soustraction	-
multiplication	*
division réelle	/
inversion de signe	-

Opérateurs de comparaison

Les deux entrées sont des **entiers**, **caractères** ou **réels**
La sortie est un **booléen**

Nom	C++
est égal à	==
est différent de	!=
est plus petit que	<
est plus grand que	>
est plus petit ou égal à	<=
est plus grand ou égal à	>=

Opérateurs logiques

Les deux entrées sont des **booléens**
La sortie est un **booléen**

Nom	C++
conjonction	&&
disjonction	||
négation	!

Expression (rappel)

Une expression est une **composition** d'opérations.

- Une expression est **bien formée** si chaque opération s'applique sur des entrées de type autorisé
- Le **type** d'une expression est donné par le type de la valeur de sortie

Expression (rappel)

Une expression est une **composition** d'opérations.

- Une expression est **bien formée** si chaque opération s'applique sur des entrées de type autorisé
- Le **type** d'une expression est donné par le type de la valeur de sortie

Exemples

Supposons que i, j et k soient des entiers

Pseudo-code	C++	Type
$(i < j)$ et $(j < k)$	`(i < j) && (j < k)`	booléen
$(i \leq j)$ ou $(j \geq k)$	`(i <= j) || (j >= k)`	booléen
$(i + j)$ mod k	`(i + j) % k`	entier

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout ≪ "Hello World" ≪ endl;
return 0;
}
```

Commentaire expliquant ce que fait ce programme

helloWorld.cpp

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout ≪ "Hello World" ≪ endl;
return 0;
}
```

Indiquer au compilateur d'inclure les fonctions entrées-sorties

helloWorld.cpp

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout ≪ "Hello World" ≪ endl;
return 0;
}
```

helloWorld.cpp

Indiquer au compilateur que l'espace des noms utilisés est celui de la bibliothèque C++ standard

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;
int main() <-- Fonction principale du programme
{
cout << "Hello World" << endl;
return 0;
}
```

helloWorld.cpp

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout << "Hello World" << endl;
return 0;
}
```

Bloc d'instructions de la fonction principale

helloWorld.cpp

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par `#`, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout ≪ "Hello World" ≪ endl;
return 0;
}
```

Afficher *Hello World* avec retour chariot

helloWorld.cpp

Structure d'un Programme C++

Le code source d'un programme C++ contient trois éléments essentiels

- **Commentaires :** préfixés par `//`, ils sont utilisés pour expliquer ce que fait le programme
- **Macros :** préfixées par #, elles sont utilisées par le pré-processeur pour manipuler le code source
- **Fonction principale :** `main()`, elle est exécutée au démarrage du programme

Exemple

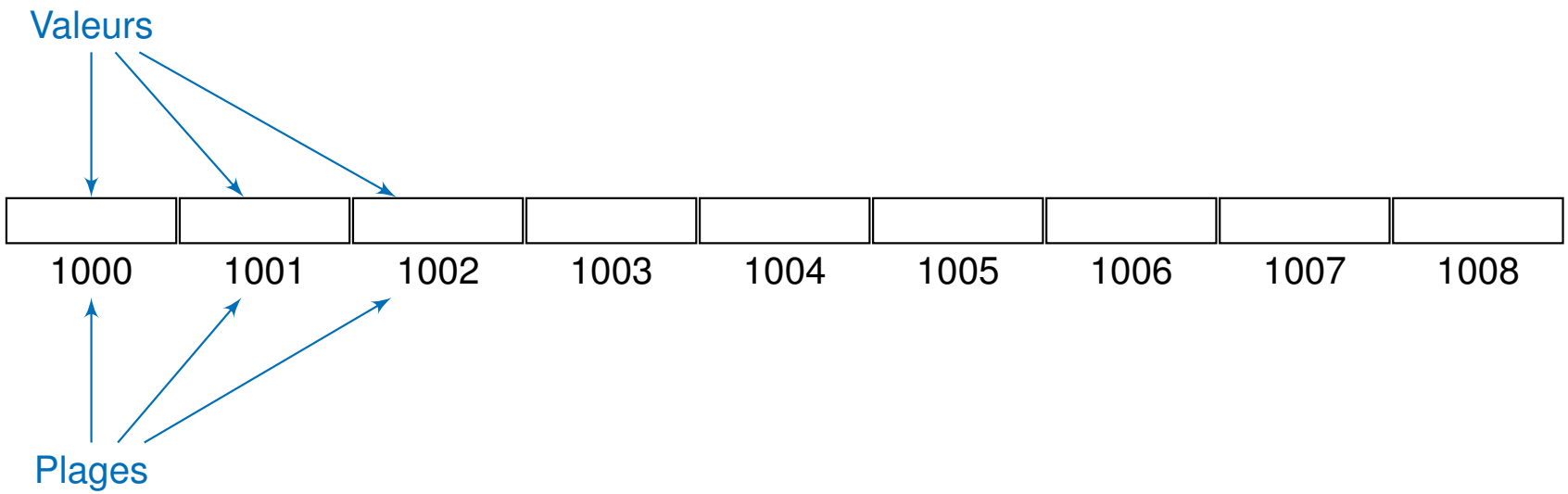
```
// Bonjour tout le monde
#include <iostream>
using namespace std;

int main()
{
cout ≪ "Hello World" ≪ endl;
return 0;
}
```

helloWorld.cpp

Valeur retournée par la fonction `main`. La valeur 0 indique que tout s'est déroulé normalement.

Mémoire

La mémoire d'un ordinateur est un tableau d'octets. Donc l'octet est la plus petite plage de mémoire que nous pouvons réserver en C++.



Instructions simples (rappel)

Une instruction simple est :

- une **déclaration** ou
- une **affectation** ou
- une **entrée-sortie**

En C++, toute instruction simple se termine par un point virgule.

Instructions simples (rappel)

Une instruction simple est :

- une **déclaration** ou
- une **affectation** ou
- une **entrée-sortie**

En C++, toute instruction simple se termine par un point virgule.

Déclaration de variable

type variable ;

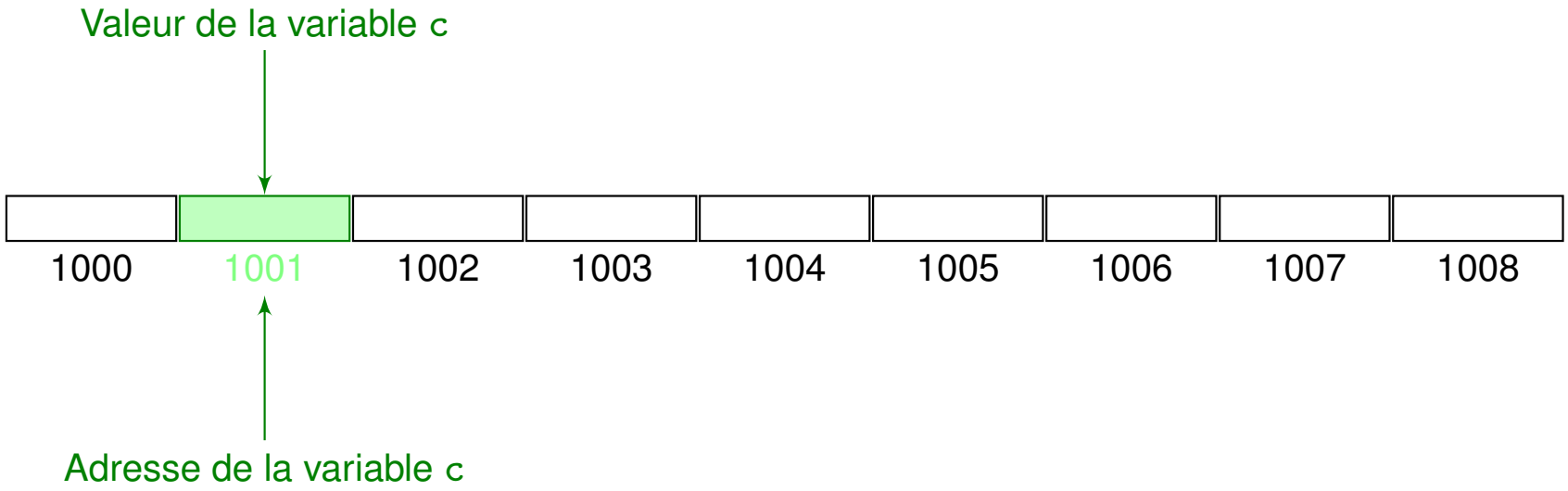
Déclaration de constante

type constante = *valeur* ;

Exemple de déclaration de variable

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

int main()
{
char c;
float f;
return 0;
}
```

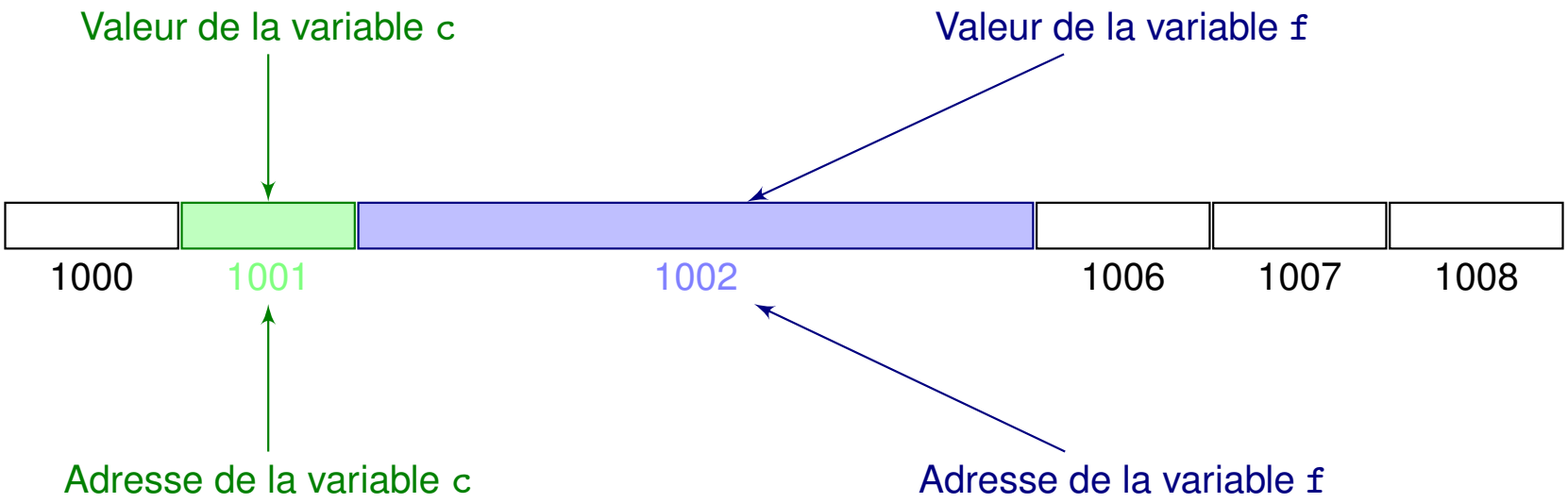
Déclaration d'une variable de type caractère



Exemple de déclaration de variable

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

int main()
{
char c;
float f;
return 0;
}
```

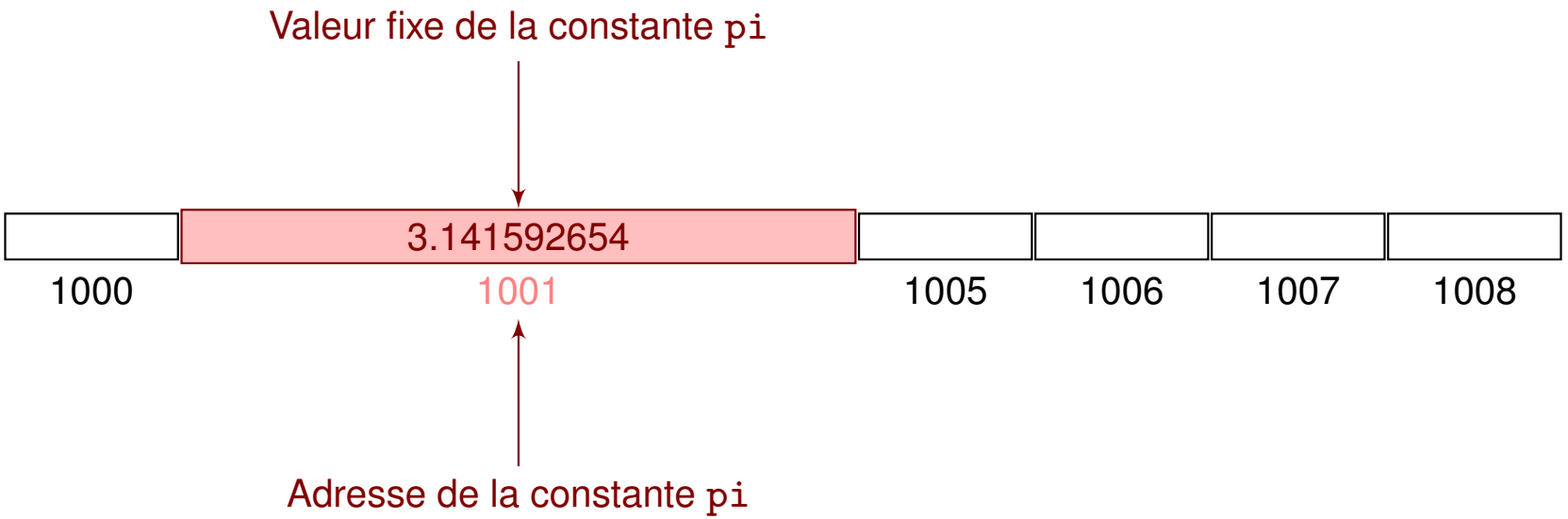
Déclaration d'une variable de type flottant



Exemple de déclaration de constante

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

int main()
{
const float pi = 3.141592654;
return 0;
}
```

Déclaration d'une constante de type flottant



Affectation de valeur

Dans une instruction de type :

variable = *expression* ;

l'opérateur = affecte la valeur de l'argument de droite (expression) à l'argument de gauche (variable). L'opérateur d'affectation (=) est différent de l'opérateur d'égalité (==) !

Exemple d'affectation (entier)

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

int main()
{
int x;
x = 10;
return 0;
}
```

Déclaration d'une variable de type entier

Affectation de valeur

Dans une instruction de type :

variable = *expression* ;

l'opérateur = affecte la valeur de l'argument de droite (expression) à l'argument de gauche (variable). L'opérateur d'affectation (=) est différent de l'opérateur d'égalité (==) !

Exemple d'affectation (entier)

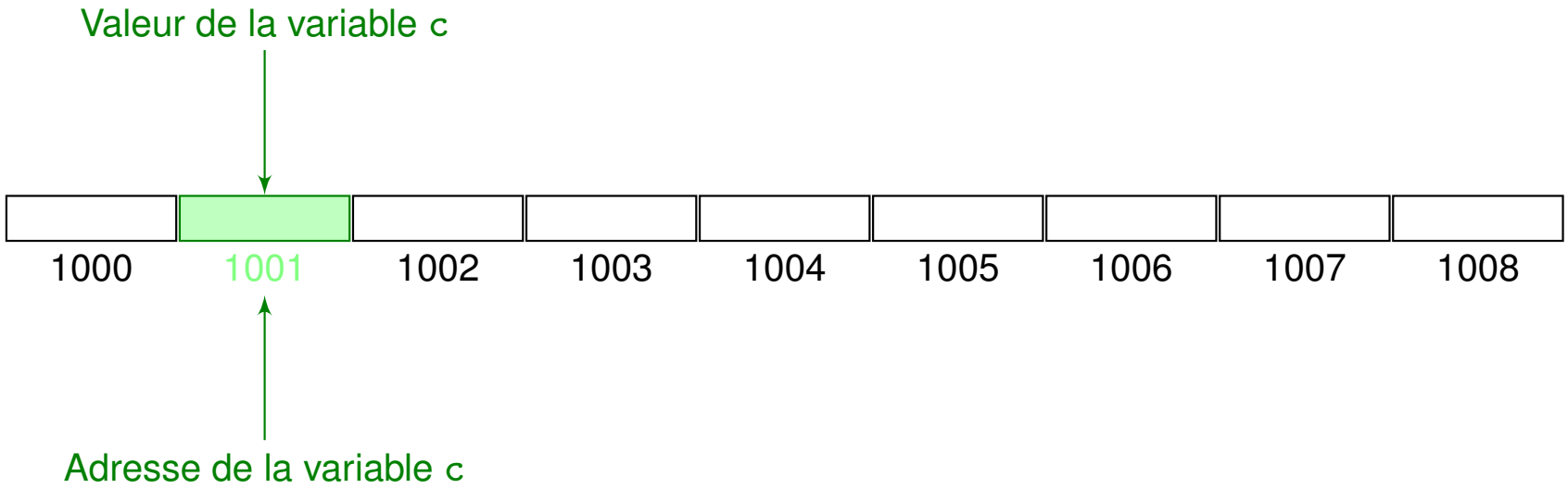
```
#include <iostream>
using namespace std;

int main()
{
int x;
x = 10;
return 0;
}
```

Affectation de la valeur 10 à `x`

Exemple d'affectation (caractère)

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

int main()
{
char c;
c = 't';
return 0;
}
```

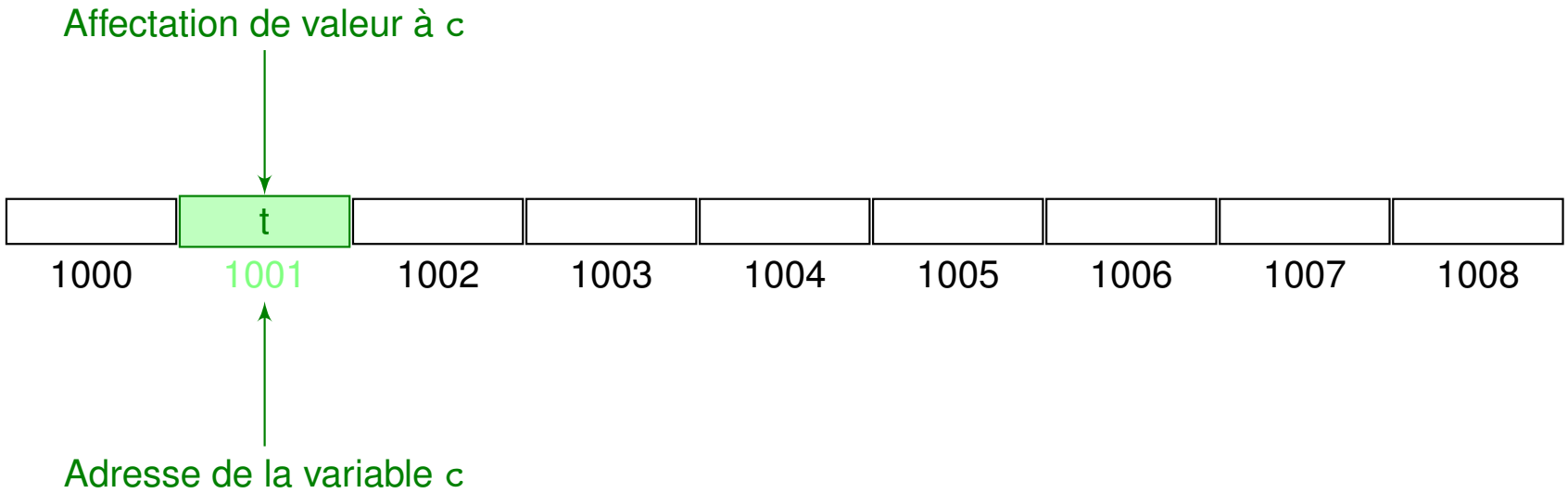
Déclaration d'une variable de type caractère



Exemple d'affectation (caractère)

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

int main()
{
char c;
c = 't';
return 0;
}
```

Affectation de la valeur t à c



Affectations combinées

Utilisées pour modifier la valeur d'une variable en effectuant une opération dessus

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

int main()
{
int x = 0;
x += 2;
return 0;
}
```

Déclaration d'une variable *x* initialisée à 0

Affectations combinées

Utilisées pour modifier la valeur d'une variable en effectuant une opération dessus

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

int main()
{
int x = 0;
x += 2;
return 0;
}
```

← La valeur de *x* est augmentée de 2

Affectation	Equivalence
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Affectations combinées

Utilisées pour modifier la valeur d'une variable en effectuant une opération dessus

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

int main()
{
int x = 0;
x += 2;
return 0;
}
```

← La valeur de *x* est augmentée de 2

Affectation	Equivalence
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Affectation	Equivalence
`x ++`	`x = x + 1`
`x --`	`x = x - 1`

Affichage

Dans une instruction de la forme :

flot ≪ *données* ;

l'opérateur ≪ envoie les données dans l'objet "flot". Par défaut, la sortie standard d'un ordinateur est l'écran et l'objet flot permettant d'y accéder est `cout`.

Exemple

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

int main()
{
...
cout ≪ "Le résultat est: "; ⟵ Affichage de la chaîne Le résultat est :
cout ≪ x ;
cout ≪ endl;
return 0;
}
```

Affichage

Dans une instruction de la forme :

flot ≪ *données* ;

l'opérateur ≪ envoie les données dans l'objet "flot". Par défaut, la sortie standard d'un ordinateur est l'écran et l'objet flot permettant d'y accéder est `cout`.

Exemple

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

int main()
{
...
cout ≪ "Le résultat est: ";
cout ≪ x ;
cout ≪ endl;
return 0;
}
```

Affichage de la valeur de la variable *x*

Affichage

Dans une instruction de la forme :

flot ≪ *données* ;

l'opérateur ≪ envoie les données dans l'objet "flot". Par défaut, la sortie standard d'un ordinateur est l'écran et l'objet flot permettant d'y accéder est `cout`.

Exemple

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

int main()
{
...
cout ≪ "Le résultat est: ";
cout ≪ x ;
cout ≪ endl;
return 0;
}
```

Retour chariot et nettoyage du tampon de flux

Lecture

Dans une instruction de la forme :

flot ≫ *données* ;

l'opérateur ≫ reçoit les données depuis l'objet "flot". Par défaut, l'entrée standard d'un ordinateur est le clavier et l'objet flot permettant d'y accéder est `cin`.

Exemple

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

int main()
{
int x;
cout ≪ "Entrez un entier : ";
cin ≫ x ; ←
...
return 0;
}
```

Lecture de la valeur de la variable *x*

Instructions composées (rappel)

Une instruction composée est :

- un **bloc**, ou
- un **test**, ou
- une **boucle**

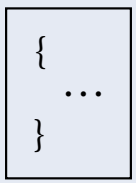
Instructions composées (rappel)

Une instruction composée est :

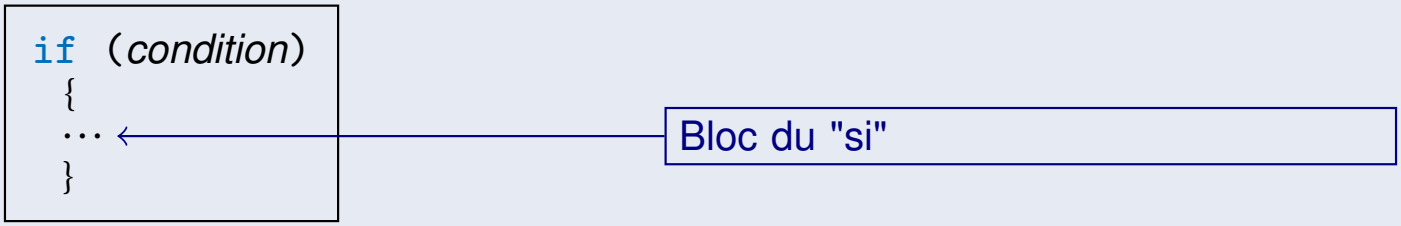
- un **bloc**, ou
- un **test**, ou
- une **boucle**

Blocs

Un bloc est un séquence d'instructions rassemblées entre crochets :

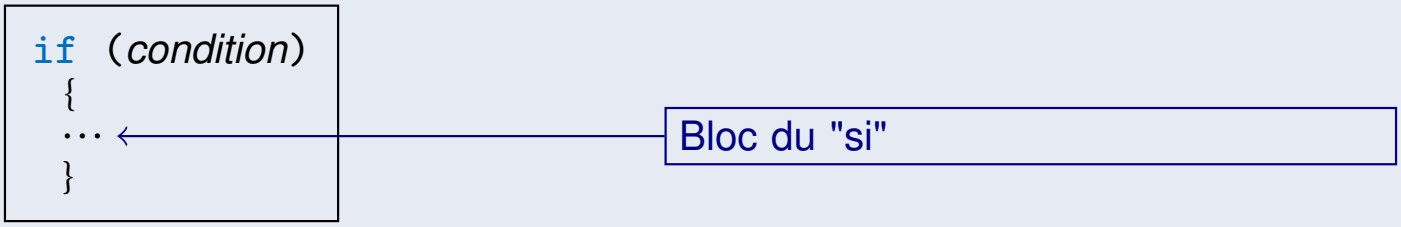


Le test “if”



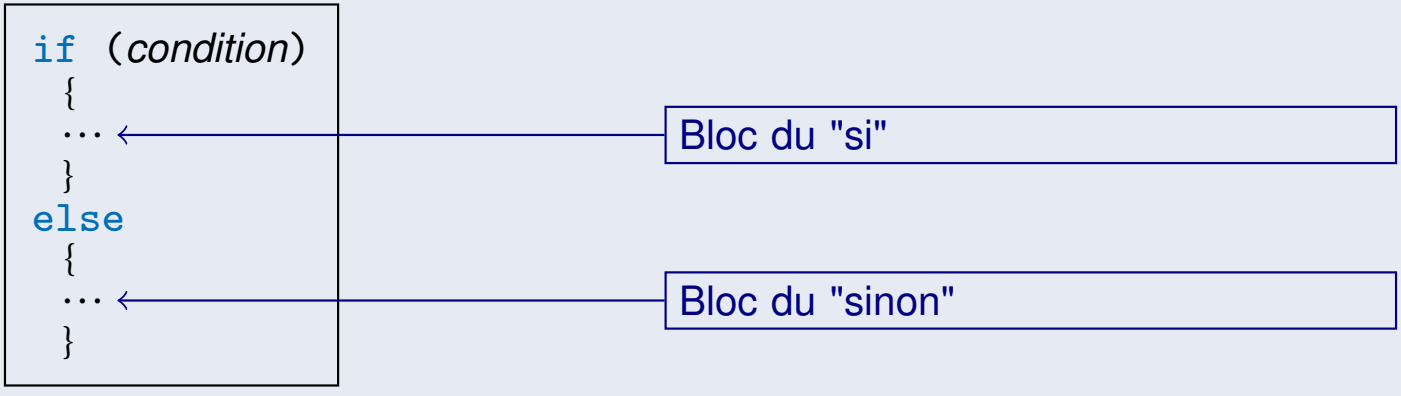
Si la *condition* est vraie, alors le bloc du "si" est exécuté. Si la condition est fausse alors le programme saute le bloc du "si" et continue après l’accolade }.

Le test “if”



Si la *condition* est vraie, alors le bloc du "si" est exécuté. Si la condition est fausse alors le programme saute le bloc du "si" et continue après l’accolade }.

Le test “if-else”



Si la *condition* est vraie, alors le bloc du "si" est exécuté. Si la *condition* est fausse, alors le bloc du "sinon" est exécuté.

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;

if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";

return 0;
}
```

Supposons que $x = 2$

RacineCarree.cpp

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;
if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";

return 0;
}
```

Supposons que $x = 2$

La condition est **vraie**

RacineCarree.cpp

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;
if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";

return 0;
}
```

RacineCarree.cpp

Supposons que $x = 2$

La condition est **vraie**

La valeur 1.414 est affichée

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;

if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";

return 0;
}
```

Supposons que $x = -1$

RacineCarree.cpp

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;
if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";

return 0;
}
```

Supposons que $x = -1$

La condition est **fausse**

RacineCarree.cpp

Exemple d'un test "if-else"

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

int main()
{
float x;

cout ≪ "Entrez un reel : ";
cin ≫ x ;
if (x >= 0)
    cout ≪ "La racine est : " ≪ sqrt(x);
else
    cout ≪ "Valeur indefinie";
return 0;
}
```

RacineCarree.cpp

Supposons que $x = -1$

La condition est **fausse**

Le message `Valeur indéfinie` est affiché

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

Le programme évalue l’expression

L’*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l’instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

si la valeur de l'expression est égale à `constante1`

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;
 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

si la valeur de l'expression est égale à `constante1`

le programme exécute la série d'instructions 1 jusqu'à ce qu'il rencontre `break`, et sort alors du test

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

si la valeur de l'expression est égale à `constante2`

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

si la valeur de l'expression est égale à `constante2`

le programme exécute la série d'instructions 2 jusqu'à ce qu'il rencontre `break`, et sort alors du test

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
  break;

 :
 default:
  groupe d'instructions par défaut
     :
 }
```

si la valeur de l'expression n'est égale à aucune des constantes précédentes

L'*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l'instruction switch.

Le test “switch-case”

```
switch (expression)
 {
 case constante1:
  groupe d'instructions 1
     :
     :
  break;
 case constante2:
  groupe d'instructions 2
     :
     :
  break;

 :
 :
 default:
  groupe d'instructions par défaut
     :
     :
 }
```

si la valeur de l’expression n’est égale à aucune des constantes précédentes

le programme exécute les instructions par défaut et sort ensuite du test

L’*expression* peut être une simple variable ou une expression arithmétique. Note : le `break` est utilisé pour faire sortir le programme de l’instruction switch.

Exemple d'un test "switch-case"

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

int main()
{
int x;
cout ≪ "Entrez un entier : ";
cin ≫ x ;


switch (x)
 {
 case 1:
 case 2:
 case 3:
   cout ≪ "x vaut 1,2 ou 3";
   break;
 default:
   cout ≪ "x ne vaut ni 1 ni 2 ni 3";
 }


return 0;
}
```

Supposons que $x = 1$

UnDeuxTrois.cpp

Exemple d'un test "switch-case"

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

int main()
{
int x;
cout ≪ "Entrez un entier : ";
cin ≫ x ;


switch (x)
 {
 case 1:
 case 2:
 case 3:
  cout ≪ "x vaut 1,2 ou 3";
  break;
 default:
  cout ≪ "x ne vaut ni 1 ni 2 ni 3";
 }


return 0;
}
```

Supposons que $x = 1$

Le programme se branche au cas 1

UnDeuxTrois.cpp

Exemple d'un test "switch-case"

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

int main()
{
int x;
cout ≪ "Entrez un entier : ";
cin ≫ x ;


switch (x)
  {
  case 1:
  case 2:
  case 3:
    cout ≪ "x vaut 1,2 ou 3";
    break;
  default:
    cout ≪ "x ne vaut ni 1 ni 2 ni 3";
  }


return 0;
}
```

Supposons que $x = 1$

Le programme se branche au cas 1

et exécute la série d'instructions jusqu'à ce qu'il rencontre `break`

UnDeuxTrois.cpp

La boucle “for”

```
for(initialisation;condition;modification)
  {
  ...
  }
```

La boucle exécute le bloc d’instructions en fonction de l’état du compteur.

- L’instruction **initialisation** affecte une valeur de départ au compteur
- L’expression **condition** teste si le bloc peut être exécuté avec la valeur courante du compteur
- L’instruction **modification** change la valeur du compteur, une fois le bloc exécuté

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout ≪ “Entrez un entier : ”;
cin ≫ n ;


for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }


cout ≪ s;
return 0;
}
```

Supposons que $n = 3$

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout ≪ “Entrez un entier : ”;
cin ≫ n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout ≪ s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i est initialisé à 1

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout ≪ “Entrez un entier : ”;
cin ≫ n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout ≪ s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i est initialisé à 1

La variable s prend la valeur $0 + 1 = 1$

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout << “Entrez un entier : ”;
cin >> n ;


for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }


cout << s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 2 ; il est plus petit ou égal à 3

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout << “Entrez un entier : ”;
cin >> n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout << s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 2 ; il est plus petit ou égal à 3

La variable s prend la valeur $1 + 2 = 3$

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout ≪ “Entrez un entier : ”;
cin ≫ n ;


for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }


cout ≪ s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 3 ; il est plus petit ou égal à 3

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout << “Entrez un entier : ”;
cin >> n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout << s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 3 ; il est plus petit ou égal à 3

La variable s prend la valeur $3 + 3 = 6$

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout ≪ “Entrez un entier : ”;
cin ≫ n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout ≪ s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 4 ; **il est plus grand que** 3

Somme de 1 à n

Exemple de boucle “for” (incrémentation)

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

int main()
{
int i,n,s;
s = 0;
cout << “Entrez un entier : ”;
cin >> n ;

for (i = 1; i <= n ; i++)
  {
  s = s + i;
  }

cout << s;
return 0;
}
```

Supposons que $n = 3$

Le compteur i passe à 4 ; **il est plus grand que** 3

La valeur 6 est affichée

Somme de 1 à n

Exemple de boucle “for” (décrémentation)

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

int main()
{
int i;


for (i = 10; i > 0 ; i--)
  {
  cout << i << « ", ";
  }


cout << "feu !" << endl;
return 0;
}
```

Compte à rebours

La boucle “while”

```
while (condition)
  {
  ...
  }
```

A chaque itération d’une boucle “while”, le programme teste la condition puis exécute le bloc si cette condition est vraie. Le bloc peut donc être exécuté zero fois.

La boucle “while”

```
while (condition)
  {
  ...
  }
```

A chaque itération d’une boucle “while”, le programme teste la condition puis exécute le bloc si cette condition est vraie. Le bloc peut donc être exécuté zero fois.

La boucle “do-while”

```
do
  {
  ...
  }
while (condition);
```

A chaque itération d’une boucle “do-while”, le programme exécute le bloc puis vérifie que la condition reste vraie. Le bloc est donc exécuté au moins une fois.

Exemple de boucle “do-while”

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

int main()
{
int iSecret, iDevine;


srand(time(NULL));
iSecret = (rand() % 10) + 1;


do
  {
  cout << "Donne un nombre de 1 à 10 : ";
  cin >> iDevine;
  if (iSecret < iDevine);
    cout << "Plus petit !" « endl;
  else if (iSecret > iDevine);
    cout << "Plus grand !" « endl;
  }
while (iSecret != iDevine);


cout << "Trouvé !" << endl;
return 0;
}
```

Génération d’un nombre aléatoire entre 1 et 10

La boucle est exécutée jusqu’à ce l’utilisateur trouve le nombre