

# Langages de programmation, structures de contrôle

A. Mémoire et variables .....	2
B. Structures de contrôle .....	19
Rôle des structures de contrôle	
Instructions conditionnelles	
Instructions itératives	

## Résumé des épisodes précédents . . .

- Informatique : Usage, Technologie, Science
- Objectif d'Info 111 : initier à la science via la technologie

## Résumé des épisodes précédents ...

- Informatique : Usage, Technologie, Science
- Objectif d'Info 111 : initier à la science via la technologie
- Concrètement : bases de la programmation impérative + ...
- Premiers programmes

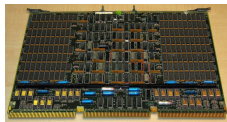
## A. Mémoire et variables

Un ordinateur traite de l'information.

- Il faut pouvoir la stocker : la **mémoire**
- Il faut pouvoir y accéder : les **variables**

# Mémoire

[https://fr.wikipedia.org/wiki/Mémoire\\_\(informatique\)](https://fr.wikipedia.org/wiki/Mémoire_(informatique))



# Mémoire

[https://fr.wikipedia.org/wiki/Mémoire\\_\(informatique\)](https://fr.wikipedia.org/wiki/Mémoire_(informatique))



## Modèle simplifié

- Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)

# Mémoire

[https://fr.wikipedia.org/wiki/Mémoire\\_\(informatique\)](https://fr.wikipedia.org/wiki/Mémoire_(informatique))



## Modèle simplifié

- Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)
- Pour 1Go, à raison de un bit par mm, cela ferait

# Mémoire

[https://fr.wikipedia.org/wiki/Mémoire\\_\(informatique\)](https://fr.wikipedia.org/wiki/Mémoire_(informatique))



## Modèle simplifié

- Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)
- Pour 1Go, à raison de un bit par mm, cela ferait 8590 km  
Plus que Paris-Pékin !



# Mémoire

[https://fr.wikipedia.org/wiki/Mémoire\\_\(informatique\)](https://fr.wikipedia.org/wiki/Mémoire_(informatique))



## Modèle simplifié

- Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)
- Pour 1Go, à raison de un bit par mm, cela ferait 8590 km  
Plus que Paris-Pékin !
- Le processeur y accède par *adresse*

# Variables

## Définition

Une *variable* est un espace de stockage **nommé** où le programme peut mémoriser une donnée ; elle possède quatre propriétés :

# Variables

## Définition

Une *variable* est un espace de stockage **nommé** où le programme peut mémoriser une donnée ; elle possède quatre propriétés :

- Un *nom* (ou *identificateur*) :  
Il est choisi par le programmeur

# Variables

## Définition

Une *variable* est un espace de stockage **nommé** où le programme peut mémoriser une donnée ; elle possède quatre propriétés :

- Un *nom* (ou *identificateur*) :  
Il est choisi par le programmeur
- Une *adresse* :  
Où est stockée la variable dans la mémoire

# Variables

## Définition

Une *variable* est un espace de stockage **nommé** où le programme peut mémoriser une donnée ; elle possède quatre propriétés :

- Un *nom* (ou *identificateur*) :  
Il est choisi par le programmeur
- Une *adresse* :  
Où est stockée la variable dans la mémoire
- Un *type* qui spécifie :
  - La *structure de donnée* : comment la valeur est représentée en mémoire  
En particulier combien d'octets sont occupés par la variable
  - La *sémantique* des opérations

# Variables

## Définition

Une *variable* est un espace de stockage **nommé** où le programme peut mémoriser une donnée ; elle possède quatre propriétés :

- Un *nom* (ou *identificateur*) :  
Il est choisi par le programmeur
- Une *adresse* :  
Où est stockée la variable dans la mémoire
- Un *type* qui spécifie :
  - La *structure de donnée* : comment la valeur est représentée en mémoire  
En particulier combien d'octets sont occupés par la variable
  - La *sémantique* des opérations
- Une *valeur* :  
Elle peut changer en cours d'exécution du programme

# Règles de formation des identificateurs

Les noms des variables (ainsi que les noms des programmes, constantes, types, procédures et fonctions) sont appelés des **identificateurs**.

## Syntaxe (règles de formation des identificateurs)

- suite de lettres (minuscules 'a'... 'z' ou majuscules 'A'... 'Z'), de chiffres ('0'... '9') et de caractères de soulignement ('\_')
- premier caractère devant être une lettre
- longueur bornée

# Règles de formation des identificateurs

Les noms des variables (ainsi que les noms des programmes, constantes, types, procédures et fonctions) sont appelés des **identificateurs**.

## Syntaxe (règles de formation des identificateurs)

- suite de lettres (minuscules 'a'... 'z' ou majuscules 'A'... 'Z'), de chiffres ('0'... '9') et de caractères de soulignement ('\_')
- premier caractère devant être une lettre
- longueur bornée

## Exemples et contres exemples d'identificateurs

- c14\_T0 est un identificateur
- 14c\_T0 n'est pas un identificateur
- x\*y n'est pas un identificateur



# Formation des identificateurs (2)

## Notes

- Donnez des noms **signifiants** aux variables
- Dans le cas de plusieurs mots, par convention dans le cadre de ce cours on mettra le premier mot en minuscule et les suivants avec une majuscule : `maVariable`
- Autre convention possible : `ma_variable`
- Mauvais noms : `truc`, `toto`, `temp`, `nombre`
- Bons noms courts : `i`, `j`, `k`, `x`, `y`, `z`, `t`
- Bons noms longs : `nbCases`, `notes`, `moyenneNotes`, `estNegatif`

# Initialisation des variables

Quelle est la valeur de ces variables après leur déclaration ?

[non-initialisation.cpp](#)

```
double d;  
long l;  
int i;
```

# Initialisation des variables

Quelle est la valeur de ces variables après leur déclaration ?

[non-initialisation.cpp](#)

```
double d;  
long l;  
int i;
```

- Certains langages ou compilateurs garantissent que les variables sont initialisées à une valeur par défaut.
- **En C++, pas forcément !**  
Typiquement, la valeur de la variable correspond à l'état de la mémoire au moment de sa déclaration

# Initialisation des variables

Quelle est la valeur de ces variables après leur déclaration ?

[non-initialisation.cpp](#)

```
double d;  
long l;  
int i;
```

- Certains langages ou compilateurs garantissent que les variables sont initialisées à une valeur par défaut.
- **En C++, pas forcément !**  
Typiquement, la valeur de la variable correspond à l'état de la mémoire au moment de sa déclaration

## Bonne pratique

Systématiquement initialiser les variables au moment de leur déclaration :

[initialisation.cpp](#)

```
int i = 0;  
long l = 1024;  
double d = 3.14159;
```

## B. Structures de contrôle

### Rappel

Les instructions sont exécutées de manière séquentielle (les unes après les autres), dans l'ordre du programme.

## B. Structures de contrôle

### Rappel

Les instructions sont exécutées de manière séquentielle (les unes après les autres), dans l'ordre du programme.

### Exemple

```
droite();  
avance();  
prend();  
gauche();  
avance();  
pose();  
droite();  
avance();  
gauche();  
avance();  
avance();  
droite();  
ouvre();
```

# Le problème

On a souvent besoin de **rompre l'exécution séquentielle** :

# Le problème

On a souvent besoin de **rompre l'exécution séquentielle** :

- Des instructions différentes, selon le contexte :



*Instructions conditionnelles*



# Le problème

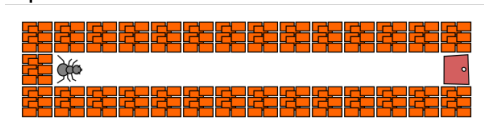
On a souvent besoin de **rompre l'exécution séquentielle** :

- Des instructions différentes, selon le contexte :



## *Instructions conditionnelles*

- Des instructions répétées :



## *Instructions itératives*

# Le problème

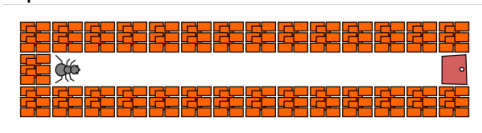
On a souvent besoin de **rompre l'exécution séquentielle** :

- Des instructions différentes, selon le contexte :



## *Instructions conditionnelles*

- Des instructions répétées :



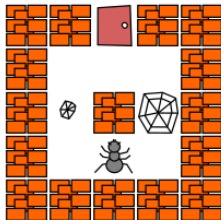
## *Instructions itératives*

Ce sont les *structures de contrôle*

## B.2. Instructions conditionnelles

En fonction d'un choix ou d'une *condition*, on va exécuter ou non un *bloc d'instructions*.

laby3a.cpp



```
droite();
avance();
gauche();

if ( regarde() == Toile ) {
    gauche();
    avance();
    avance();
    droite();
    avance();
    droite();
    avance();
    gauche();
} else {
    avance();
    avance();
    gauche();
    avance();
    droite();
}
ouvre();
```

# Expressions booléennes, blocs d'instructions

## Définition

Une *condition* est une *expression booléenne*

*i.e.* dont le résultat est de type booléen : vrai (`true`) ou faux (`false`)

# Expressions booléennes, blocs d'instructions

## Définition

Une *condition* est une *expression booléenne*

i.e. dont le résultat est de type booléen : vrai (**true**) ou faux (**false**)

## Exemples

regarde() == Toile

x > 3.14

2 <= n **and** n <= 5

# Expressions booléennes, blocs d'instructions

## Définition

Une *condition* est une *expression booléenne*  
i.e. dont le résultat est de type booléen : vrai (**true**) ou faux (**false**)

## Exemples

regarde() == Toile            x > 3.14            2 <= n **and** n <= 5

## Définition

Un *bloc* d'instructions est une suite d'instructions à exécuter successivement. Il est décrit par la syntaxe suivante :

```
{  
    instruction 1;  
    instruction 2;  
    ...  
    instruction n;  
}
```

# Expressions booléennes, blocs d'instructions

## Définition

Une *condition* est une *expression booléenne*  
i.e. dont le résultat est de type booléen : vrai (**true**) ou faux (**false**)

## Exemples

regarde() == Toile            x > 3.14            2 <= n **and** n <= 5

## Définition

Un *bloc* d'instructions est une suite d'instructions à exécuter successivement. Il est décrit par la syntaxe suivante :

```
{  
    instruction 1;  
    instruction 2;  
    ...  
    instruction n;  
}
```

Une instruction toute seule est considérée comme un bloc

# Instruction conditionnelle simple : « si ... alors ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions;  
}
```



# Instruction conditionnelle simple : « si ... alors ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions;  
}
```

## Sémantique

1. Évaluation de la condition
2. Si sa valeur est **true**, exécution du bloc d'instructions

# Instruction conditionnelle simple : « si ... alors ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions;  
}
```

## Sémantique

1. Évaluation de la condition
2. Si sa valeur est **true**, exécution du bloc d'instructions

## Exemples

```
if ( regarde() == Toile ) {      // Au secours, fuyons!  
    gauche();  
    gauche();  
}
```

# Instruction conditionnelle simple : « si ... alors ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions;  
}
```

## Sémantique

1. Évaluation de la condition
2. Si sa valeur est **true**, exécution du bloc d'instructions

## Exemples

```
if ( regarde() == Toile ) {           // Au secours, fuyons!  
    gauche();  
    gauche();  
}
```

```
if ( x >= 0 ) gauche();
```

# Instruction conditionnelle : « si ... alors ... sinon ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions 1;  
} else {  
    bloc d'instructions 2;  
}
```

# Instruction conditionnelle : « si ... alors ... sinon ... »

## Syntaxe

```
if ( condition ) {  
    bloc d'instructions 1;  
} else {  
    bloc d'instructions 2;  
}
```

## Sémantique

1. Évaluation de la condition
2. Si sa valeur est **true**, exécution du bloc d'instructions 1
3. Si sa valeur est **false**, exécution du bloc d'instructions 2

# Exemples d'instruction alternative

## Exemple

```
if ( regarde() == Toile ) {      // Au secours, fuyons!  
    gauche();  
    gauche();  
} else {                        // Tout va bien  
    avance();  
}
```

## Exemples d'instruction alternative (2)

Exemple (Calcul du maximum et du minimum de x et y)

```
int x, y;                // Les entrées
int maximum, minimum;    // Les sorties

if ( x > y ) {
    maximum = x;
    minimum = y
} else {
    maximum = y;
    minimum = x;
}
```

# Erreurs classiques avec les conditionnelles

## Exemple

```
bool estPositif;  
if ( x >= 0 ) {  
    estPositif = true  
} else {  
    estPositif = false  
}
```



# Erreurs classiques avec les conditionnelles

## Exemple

```
bool estPositif;  
if ( x >= 0 ) {  
    estPositif = true  
} else {  
    estPositif = false  
}
```

Utiliser une expression booléenne à la place !

```
bool estPositif = x >= 0;
```

## Erreurs classiques avec les conditionnelles (2)

### Exercice

Que fait :

```
if ( x = 1 ) {  
    y = 4;  
}
```

## Erreurs classiques avec les conditionnelles (2)

### Exercice

Que fait :

```
if ( x = 1 ) {  
    y = 4;  
}
```

Attention !

Ne pas confondre « = » (affectation) et « == » (égalité) !

## Erreurs classiques avec les conditionnelles (3)

### Exercice

Que fait :

```
if ( x == 1 ); {  
    y = 4;  
}
```

## Erreurs classiques avec les conditionnelles (3)

### Exercice

Que fait :

```
if ( x == 1 ); {  
    y = 4;  
}
```

La même chose que :

```
if ( x == 1 );  
y = 4;
```

Ne tiens pas compte du `if` et affiche toujours « x vaut 1 ».

## Erreurs classiques avec les conditionnelles (3)

### Exercice

Que fait :

```
if ( x == 1 ); {  
    y = 4;  
}
```

La même chose que :

```
if ( x == 1 );  
y = 4;
```

Ne tiens pas compte du `if` et affiche toujours « x vaut 1 ».

### Attention !

- le point-virgule est un séparateur d'instruction !
- `if (...) {...} else {...}` forme une seule instruction
- **Jamais de point-virgule avant un bloc d'instructions !**

# Tests imbriqués

## Exemple

Que se passe-t-il lorsque  $x == 5$  et  $y == 4$  dans l'exemple suivant :

```
if ( x >= y ) {  
    if ( x == y ) {  
        resultat = "égalité";  
    }  
else {  
    resultat = "x est plus petit que y"  
}  
}
```

# Tests imbriqués

## Exemple

Que se passe-t-il lorsque  $x == 5$  et  $y == 4$  dans l'exemple suivant :

```
if ( x >= y ) {  
    if ( x == y ) {  
        resultat = "égalité";  
    }  
    else {  
        resultat = "x est plus petit que y"  
    }  
}
```



# Tests imbriqués

## Exemple

Que se passe-t-il lorsque  $x == 5$  et  $y == 4$  dans l'exemple suivant :

```
if ( x >= y ) {  
    if ( x == y ) {  
        resultat = "égalité";  
    }  
}  
else {  
    resultat = "x est plus petit que y"  
}
```

# Tests imbriqués

## Exemple

Que se passe-t-il lorsque  $x == 5$  et  $y == 4$  dans l'exemple suivant :

```
if ( x >= y ) {  
    if ( x == y ) {  
        resultat = "égalité";  
    }  
}  
else {  
    resultat = "x est plus petit que y"  
}
```

## Attention !

- **un else se rapporte au dernier if rencontré.**
- En C++, la structuration est déterminée par les accolades

# Tests imbriqués

## Exemple

Que se passe-t-il lorsque  $x == 5$  et  $y == 4$  dans l'exemple suivant :

```
if ( x >= y ) {  
    if ( x == y ) {  
        resultat = "égalité";  
    }  
}  
else {  
    resultat = "x est plus petit que y"  
}
```

## Attention !

- **un else se rapporte au dernier if rencontré.**
- En C++, la structuration est déterminée pas les accolades
- La mauvaise **indentation** induit en erreur le lecteur !

# L'indentation

« *Programs must be written for people to read, and only incidentally for machines to execute.* »

– *Harold Abelson, Structure and Interpretation of Computer Programs 1984*

## Rappel

- Un programme s'adresse à un **lecteur**
- La **lisibilité** est un objectif essentiel

# L'indentation

« *Programs must be written for people to read, and only incidentally for machines to execute.* »

– *Harold Abelson, Structure and Interpretation of Computer Programs 1984*

## Rappel

- Un programme s'adresse à un **lecteur**
- La **lisibilité** est un objectif essentiel

## Notes

- L'**indentation** consiste à espacer les lignes de code par rapport au bord gauche de la fenêtre de saisie de texte
- L'espacement doit être proportionnel au **niveau d'imbrication** des instructions du programme
- Quatre espaces par niveau d'imbrication est un bon compromis

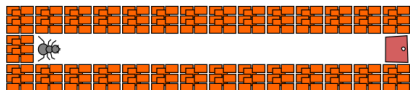
La plupart des éditeurs de texte offrent des facilités pour réaliser une bonne indentation. **Apprenez les.**



# Instructions itératives : exemple

[laby2a-mauvais.cpp](#)

```
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
avance();  
ouvre();
```



[laby2a.cpp](#)

```
while ( regarde() == Vide ) {  
    avance();  
}  
ouvre();
```

## B.3. Instructions itératives

### Rappel

La force d'un ordinateur est de savoir faire des tâches répétitives très rapidement et sans s'ennuyer



## B. 3. Instructions itératives

### Rappel

La force d'un ordinateur est de savoir faire des tâches répétitives très rapidement et sans s'ennuyer

### Exemples

- On veut afficher tous les nombres entre 1 et 1000.
- Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.
- Tous les  $1/24^{\text{ème}}$  de seconde on veut afficher une image d'un film (s'il en reste)

# Les instructions itératives

## Définition

Les *instructions itératives* permettent de répéter un certain nombre de fois l'exécution d'un bloc d'instructions sous certaines conditions

# Les instructions itératives

## Définition

Les *instructions itératives* permettent de répéter un certain nombre de fois l'exécution d'un bloc d'instructions sous certaines conditions

De façon imagée, on appelle **boucle** cette méthode permettant de répéter l'exécution d'un groupe d'instructions.

# Les instructions itératives

## Définition

Les *instructions itératives* permettent de répéter un certain nombre de fois l'exécution d'un bloc d'instructions sous certaines conditions

De façon imagée, on appelle **boucle** cette méthode permettant de répéter l'exécution d'un groupe d'instructions.

## Instructions itératives

- Boucles while : « tant que ... faire ... »
- Boucles do ... while : « Faire ... tant que ... »
- Boucles for : « Pour ... de ... à ... faire ... »

# La boucle while : « tant que ... répéter ... »

## Syntaxe

```
while ( condition ) {  
    bloc d'instructions;  
}
```

# La boucle while : « tant que ... répéter ... »

## Syntaxe

```
while ( condition ) {  
    bloc d'instructions;  
}
```

## Sémantique

1. Évaluation de la condition
2. Si la valeur est **true** :
  - 2.1 Exécution du bloc d'instructions
  - 2.2 On recommence en 1.

# La boucle while : exemple

## Exemple (Compter de 1 à 5)

```
int n = 1;

while ( n <= 5 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

## Cas particulier : condition toujours fausse

**Si la valeur de la condition est fausse dès le départ, alors le bloc d'instructions ne sera jamais exécuté !**



## Cas particulier : condition toujours fausse

**Si la valeur de la condition est fausse dès le départ, alors le bloc d'instructions ne sera jamais exécuté !**

### Exemple

```
int n = 1;

while ( n < 0 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Cas particulier : condition toujours vraie

**Si la valeur de la condition est toujours vraie, alors le bloc d'instructions sera exécuté indéfiniment ! (boucle infinie)**

## Cas particulier : condition toujours vraie

**Si la valeur de la condition est toujours vraie, alors le bloc d'instructions sera exécuté indéfiniment ! (boucle infinie)**

Exemple (Que fait ce programme?)

```
int n = 1;

while ( true ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

## Cas particulier : condition toujours vraie

**Si la valeur de la condition est toujours vraie, alors le bloc d'instructions sera exécuté indéfiniment ! (boucle infinie)**

Exemple (Que fait ce programme?)

```
int n = 1;

while ( true ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Exemple (Erreur typique : oublier l'incrémentation !)

```
int n = 1;

while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
}
```

# Une source d'erreur classique en fin de boucle

## Exemple

Que vaut  $n$  à la fin du programme suivant ?

```
int n = 1;

while ( n <= 10 ) {
    n = n + 1;
}

cout << n << endl;    // Affiche la valeur de n
```

# Une source d'erreur classique en fin de boucle

## Exemple

Que vaut  $n$  à la fin du programme suivant ?

```
int n = 1;

while ( n <= 10 ) {
    n = n + 1;
}

cout << n << endl;    // Affiche la valeur de n
```

## Rappel

On sort de la boucle quand la condition est **false**

Le compteur est donc « un cran trop loin »

## La boucle do ... while : « faire ... tant que ... »

### Exemple

Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.

## La boucle do ... while : « faire ... tant que ... »

### Exemple

Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.

- Jouer la partie **au moins une fois**
- Tester la condition **après** la partie



# La boucle do ... while : « faire ... tant que ... »

## Exemple

Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.

- Jouer la partie **au moins une fois**
- Tester la condition **après** la partie

## Syntaxe

```
do {  
    bloc d'instructions  
} while ( condition );
```

# La boucle do ... while : « faire ... tant que ... »

## Exemple

Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.

- Jouer la partie **au moins une fois**
- Tester la condition **après** la partie

## Syntaxe

```
do {  
    bloc d'instructions  
} while ( condition );
```

## Sémantique

1. Exécution du bloc d'instructions
2. Évaluation de la condition
3. Si sa valeur est **true**, on recommence en 1.

# La boucle do ... while : exemples

## Exemple

```
char reponse;  
  
do {  
    ...  
    cout << "Voulez-vous rejouer (o/n) ?" << endl;  
    cin >> reponse;  
} while ( reponse == 'o' );
```

## La boucle for : « pour ... de ... à ... faire ... »

### Exemple (Compter de 1 à 10)

```
int n = 1;
while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

## La boucle for : « pour ... de ... à ... faire ... »

### Exemple (Compter de 1 à 10)

```
int n = 1;
while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

### Schéma classique avec un **compteur** :

```
initialisation;
while ( condition ) {
    bloc d'instructions
    incrementation
}
```

## La boucle for : « pour ... de ... à ... faire ... »

Exemple (Compter de 1 à 10)

```
int n = 1;
while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Schéma classique avec un **compteur** :

```
initialisation;
while ( condition ) {
    bloc d'instructions
    incrementation
}
```

**Gestion du compteur dispersée !**

## La boucle for : « pour ... de ... à ... faire ... »

### Exemple (Compter de 1 à 10)

```
int n = 1;
while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Avec une boucle **for** :

```
for ( int n = 1; n <= 10; n = n + 1 ) {
    cout << n << endl;    // Affiche la valeur de n
}
```

# La boucle for : « pour ... de ... à ... faire ... »

## Syntaxe

```
for ( initialisation ; condition ; incrementation ) {  
    bloc d'instructions  
}
```



# La boucle for : « pour ... de ... à ... faire ... »

## Syntaxe

```
for ( initialisation ; condition ; incrementation ) {  
    bloc d'instructions  
}
```

## Sémantique

1. Exécution de l'instruction d'initialisation
2. Évaluation de la condition
3. Si sa valeur est **true** :
  - 3.1 Exécution du bloc d'instruction
  - 3.2 Exécution de l'instruction d'incrément
  - 3.3 On recommence en 2.

# La boucle for : « pour ... de ... à ... faire ... »

## Syntaxe

```
for ( initialisation ; condition ; incrementation ) {  
    bloc d'instructions  
}
```

## Sémantique

1. Exécution de l'instruction d'initialisation
2. Évaluation de la condition
3. Si sa valeur est **true** :
  - 3.1 Exécution du bloc d'instruction
  - 3.2 Exécution de l'instruction d'incrément
  - 3.3 On recommence en 2.

## Remarque

- **Centralise** la gestion du compteur
- Strictement équivalent à while, mais exprime une **intention**

# La boucle for : exemples

## Exemple

```
int n;  
for ( n = 1 ; n <= 10 ; n = n + 1 ) {  
    cout << n;  
}
```

# La boucle for : exemples

## Exemple

```
int n;  
for ( n = 1 ; n <= 10 ; n = n + 1 ) {  
    cout << n;  
}
```

Variante compacte :

```
for ( int n = 1 ; n <= 10 ; n++ ) {  
    cout << n;  
}
```

# La boucle for : exemples

## Exemple

```
int n;  
for ( n = 1 ; n <= 10 ; n = n + 1 ) {  
    cout << n;  
}
```

Variante compacte :

```
for ( int n = 1 ; n <= 10 ; n++ ) {  
    cout << n;  
}
```

- La variable `n` est locale à la boucle (on y reviendra)
- `n++` est un raccourci pour `n = n + 1`

# La boucle for : calcul de la factorielle

## Exemple

On veut calculer  $7! = 1 \cdot 2 \cdots 7$  :

[factorielle-7.cpp](#)

```
int resultat = 1;

resultat = resultat * 2;
resultat = resultat * 3;
resultat = resultat * 4;
resultat = resultat * 5;
resultat = resultat * 6;
resultat = resultat * 7;
```

# La boucle for : calcul de la factorielle

## Exemple

On veut calculer  $7! = 1 \cdot 2 \cdots 7$  :

`factorielle-7.cpp`

```
int resultat = 1;

resultat = resultat * 2;
resultat = resultat * 3;
resultat = resultat * 4;
resultat = resultat * 5;
resultat = resultat * 6;
resultat = resultat * 7;
```

## Problèmes

- Ce code sent mauvais (répétitions) !
- Et si on veut calculer  $10!$  ou  $100!$  ?

## La boucle for : calcul de la factorielle (2)

### Exemple

Entrée : un entier  $n$

factorielle-for.cpp

```
int resultat = 1;

for ( int k = 1; k <= n; k++ ) {
    resultat = resultat * k;
}
```

Exécution pour  $n = 3$  puis pour  $n = 0$



# La boucle for : calcul de la factorielle (2)

## Exemple

Entrée : un entier  $n$

factorielle-for.cpp

```
int resultat = 1;

for ( int k = 1; k <= n; k++ ) {
    resultat = resultat * k;
}
```

Exécution pour  $n = 3$  puis pour  $n = 0$

## Techniques classiques de boucles

- Utilisation d'un *compteur* : `k`  
Varie toujours de la même façon
- Utilisation d'un *accumulateur* : `resultat`  
Accumule progressivement des valeurs par produit, somme, ...

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

## Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

## Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »
  - Erreurs classiques

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

## Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »
  - Erreurs classiques
- **Instructions itératives**
  - Boucles while : « tant que ... faire ... »
  - Boucles do while : « faire ... tant que ... »
  - Boucles for : « pour ... de ... à ... faire ... »

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

## Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »
  - Erreurs classiques
- **Instructions itératives**
  - Boucles while : « tant que ... faire ... »
  - Boucles do while : « faire ... tant que ... »
  - Boucles for : « pour ... de ... à ... faire ... »
  - Compteurs, accumulateurs
  - Erreurs classiques

# Résumé

## Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

## Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »
  - Erreurs classiques
- **Instructions itératives**
  - Boucles while : « tant que ... faire ... »
  - Boucles do while : « faire ... tant que ... »
  - Boucles for : « pour ... de ... à ... faire ... »
  - Compteurs, accumulateurs
  - Erreurs classiques

## Importance de la lisibilité du code

- indentation, ...