



Programmation Orientée Objet en C++

Filière : Ingénierie Informatique et Réseaux

ECOLE MAROCAINE DES SCIENCES DE L'INGÉNIEUR – EMSI
Honoris United Universities

Niveau : 3°IIR

Pr. NAZIH Marouane

Table of Contents

- I. Introduction au langage C++
- II. Notions de classes, constructeurs, destructeurs
- III. Surcharge des fonctions & opérateurs
- IV. Les techniques de l'héritage
- V. Fonctions & classes amies
- VI. Les fonctions virtuelles et les classes abstraites
- VII. Héritage multiple
- VIII. Polymorphisme
- IX. Les exceptions
- X. Bibliothèque SDL

Introduction au langage C++

Algorithme et programme



Algorithme et programme

□ Définition: Algorithme

Un algorithme est une suite [finie] d'opérations élémentaires permettant d'obtenir le résultat final déterminé à un problème.

□ Algorithme: méthode pour résoudre un problème.

□ Propriété d'un algorithme

Un algorithme, dans des conditions d'exécution similaires (avec des données identiques) fournit toujours le même résultat.

Algorithme et programme

□ Définition: Programme

Un programme informatique est un ensemble d'opérations destinées à être exécutées par un ordinateur.

□ Programme: s'adresse à une machine !

Algorithme et programme

| Caractéristique | Algorithme | Programme |
|-----------------|---|--|
| Définition | Séquence d'instructions pour résoudre un problème | Collection d'instructions écrites dans un langage de programmation |
| Indépendance | Indépendant du langage | Dépendant d'un langage de programmation spécifique |
| Nature | Abstrait, logique | Concret, exécutable |
| Finitude | Doit se terminer après un nombre fini d'étapes | Peut être exécuté indéfiniment ou jusqu'à ce qu'il soit arrêté |
| Exécution | Non exécutable directement | Exécutable par un ordinateur |

Langage de programmation



Langage de programmation

- Le **langage** est la capacité d'exprimer une pensée et de communiquer au moyen d'un système de signes doté d'une sémantique, et le plus souvent d'une syntaxe. Plus couramment, **le langage est un moyen de communication.**
- Un **langage de programmation** est un code de communication, permettant à un être humain de dicter des ordres (instructions) à une machine qu'elle devra exécuter.

Types de langages de programmation

❑ Langages de programmation impératifs :

Langages procéduraux : C, Pascal, et Fortran.

Langages orientés objet : Java, C++, et Python.

Langages basés sur les événements : JavaScript et Visual Basic.

❑ Langages de programmation déclaratifs :

Langages de requête : SQL pour les bases de données.

Langages de description de données : HTML et XML pour la structuration des données.

Types de langages de programmation

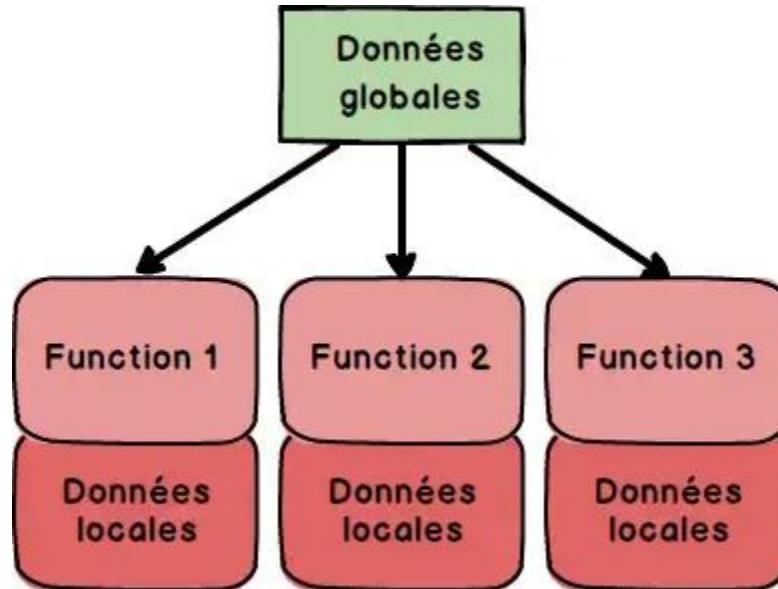
❑ Langages de script :

JavaScript, Python et Ruby. Ces langages sont souvent utilisés pour l'automatisation de tâches et le développement rapide d'applications.

❑ Langages de programmation embarqués :

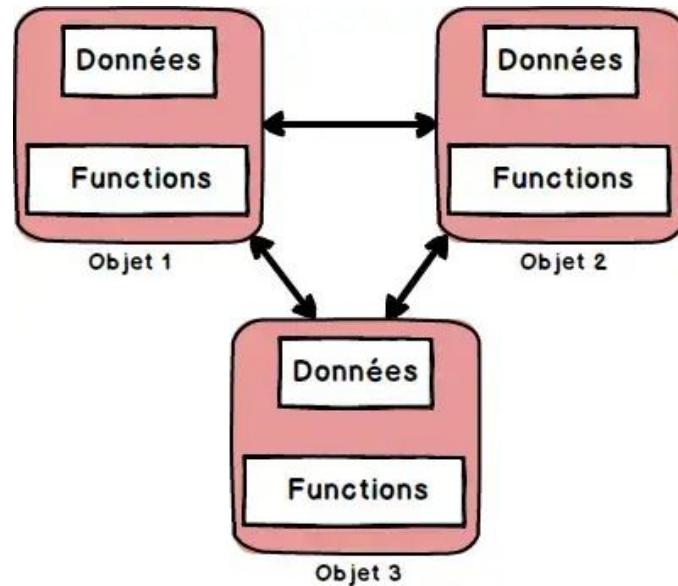
C pour la programmation de microcontrôleurs et VHDL pour la conception de circuits intégrés.

Programmation procédurale



Dans la **programmation procédurale** le programme est divisé en petites parties appelées **procédures** ou **fonctions**.

Programmation orientée objet



Dans la **programmation orientée objet** le programme est divisé en parties appelées **objets**.

Procédurale VS POO

la différence entre la programmation procédurale et la programmation orientée objet (POO) réside dans le fait que dans:

la programmation procédurale, les programmes sont basés sur des fonctions, et les données peuvent être facilement accessibles et modifiables,

alors qu'en programmation orientée objet, chaque programme est constitué d'entités appelées objets, qui ne sont pas facilement accessibles et modifiables.

HISTORIQUE

- **1972 : Langage C**
AT&T Bell Labs.
- **1979: C with classes**
Bjarne Stroustrup développe le langage *C with classes*.
- **1985 : C++**
AT&T Bell Labs; **Extension Objet** par **Bjarne Stroustrup**.
- **1995 : Java**
Sun Microsystems puis Oracle; **Simplification du C++, purement objet**, également inspiré de Smalltalk, ADA, etc.
- **1998 :C++98: Normalisation du C++ par l'ISO**
(l'Organisation internationale de normalisation).

HISTORIQUE

□ 2001: C#

Microsoft; Originellement proche de **Java** pour **.NET**, également inspiré de **C++**, Delphi, etc.

□ 2011: C++11

Révision majeure du C++

□ 2014: C++14

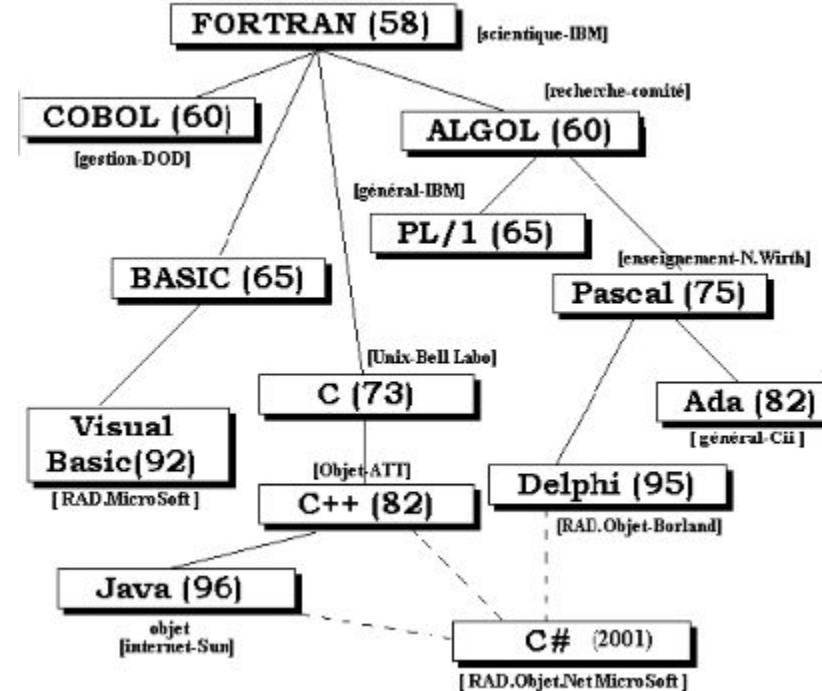
Mise à jour mineure du langage C++11

□ 2017: C++17

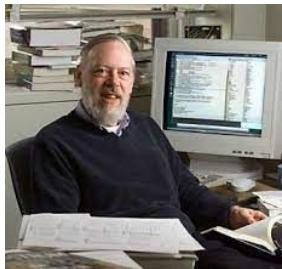
Sortie de la dernière version

□ C++20: planifié depuis juillet 2017.

HISTORIQUE



HISTORIQUE



Créateur du Langage C
Dennis Ritchie



Créateur du Langage C++
Bjarne Stroustrup

C

1972

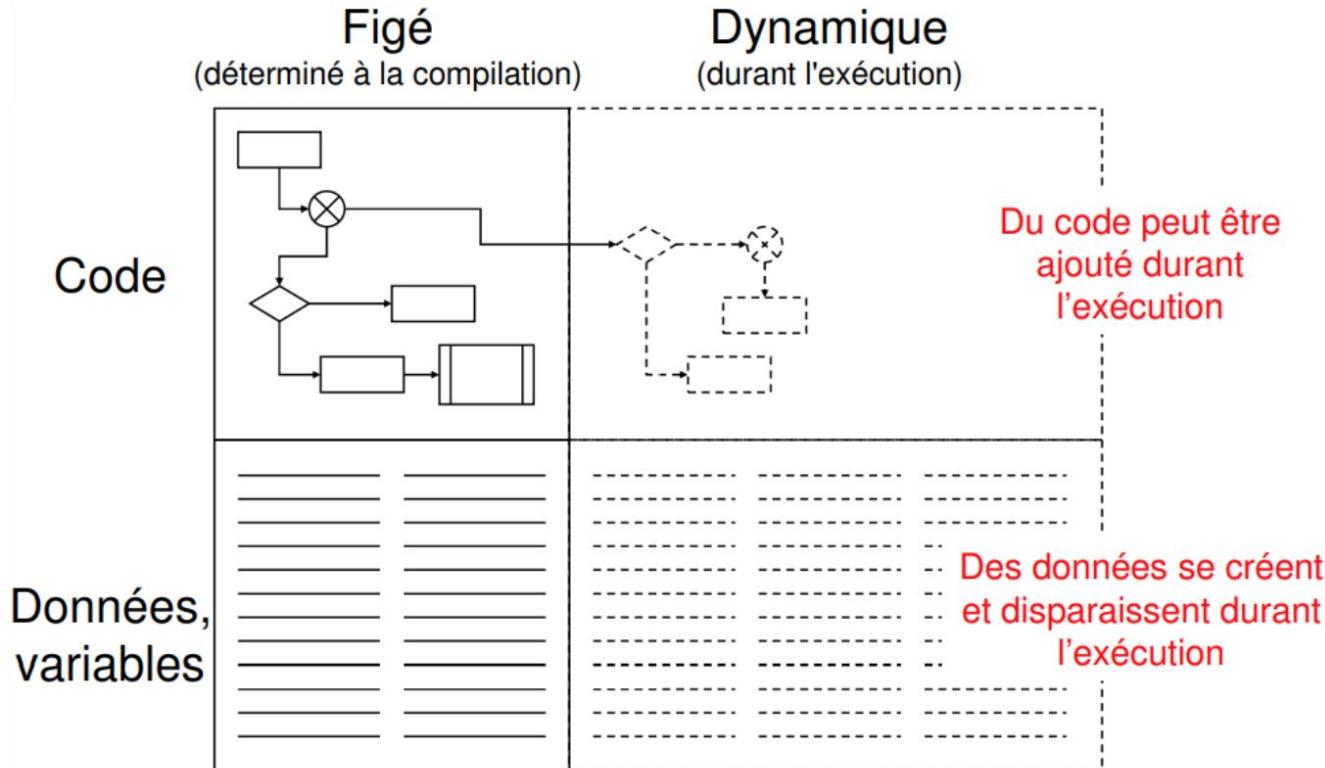
C with Class

1979

C++

1985

Typologie des langages informatiques



Typologie des langages: typage

On peut aussi distinguer les langages selon leur façon de gérer le typage des variables :

- Typage statique

Les types sont fixés à la compilation: Déclaration de toutes les variables et de leur type avant leur utilisation.

- Typage dynamique

Un langage dans lequel les types sont découverts à l'exécution: Détermination du type d'une variable la première fois qu'on lui assigne une valeur.

Exemple en C++

```
#include <stdio.h>

int main() {
    int nombre = 5; // Déclaration d'une variable de type int
    printf("Le nombre est : %d\n", nombre);

    // Tentative d'assigner une chaîne de caractères à la même
    // variable (erreur de type)
    // Cette erreur sera détectée lors de la compilation.
    nombre = "Bonjour"; // Erreur : incompatible types
    printf("Le nombre est : %d\n", nombre);

    return 0;
}
```

Exemple en Python

```
nombre = 5
print("Le nombre est :", nombre)

# Cette erreur sera détectée à l'exécution.
nombre = "Bonjour" # Pas d'erreur de type à la compilation
print("Le nombre est :", nombre)
```

Organisation d'un programme C++

- Programme C++ généralement constitué de plusieurs modules, compilés séparément
- Fichier entête – d'extension .h (ou .hh ou .hpp)
 - Contenant les déclarations de types, fonctions, variables et constantes, etc.
 - Inclus via la commande include
- Fichier source – d'extension .cpp ou .c

MonFichierEnTete.h

```
#include <iostream>
extern char* MaChaine;
extern void MaFonction();
```

MonFichier.cpp

```
#include "MonFichierEnTete.h"
void MaFonction()
{
    cout << MaChaine << " \n " ;
}
```

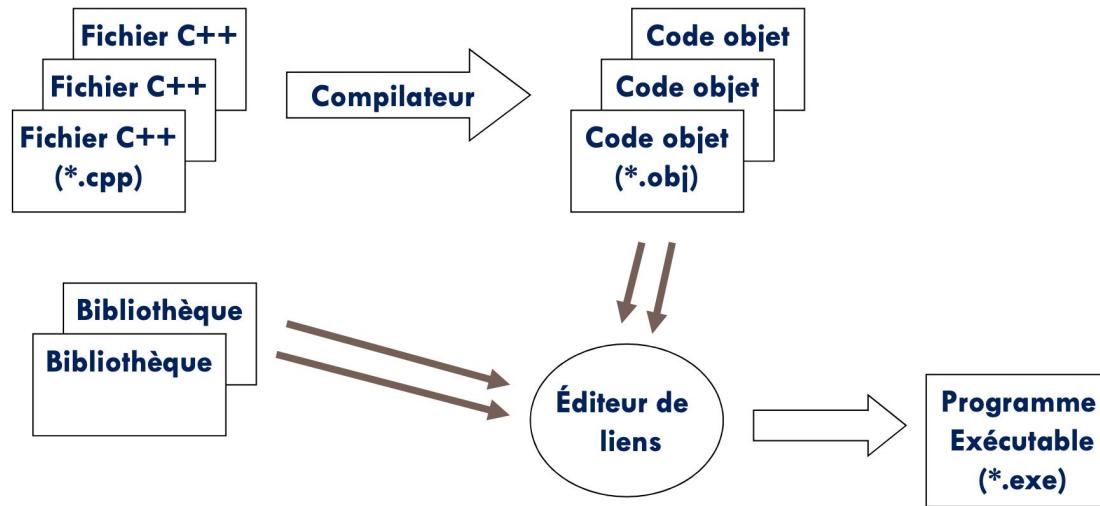
MonProgPrincipal.cpp

```
#include "MonFichierEnTete.h"

char *MaChaine="Chaîne à afficher";
int main()
{
    MaFonction();
}
```

Organisation d'un programme C++

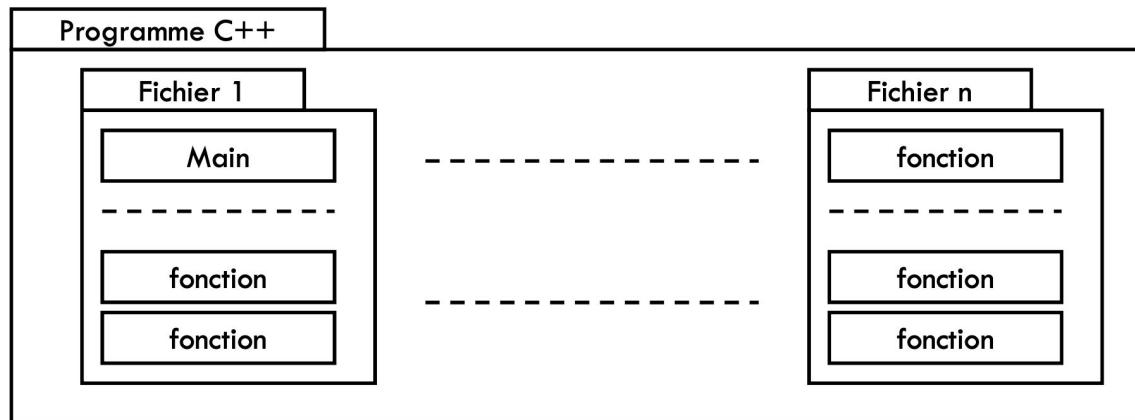
- Un programme se présente comme une suite de fichiers.



C++ est un langage compilé

Organisation d'un programme C++

- Chaque fichier se présente comme une suite de fonctions.
- Une fonction est imposée: la fonction *main()*
 - Fonction principale.
 - Point d'entrée du programme.



Compilation

- Un programme en langage C++ est un **fichier texte**, que l'on écrit à l'aide d'un **éditeur de texte**.
- Ce programme en langage C++ n'est pas exécutable directement par la machine: il doit être *compilé* pour pouvoir être exécuté par l'ordinateur.
- La compilation est réalisée par un programme appelé **compilateur**. Le compilateur crée un **fichier exécutable**.

Compilation

- Langage C++ : langage compilé ⇒ fichier exécutable produit à partir de fichiers sources par un compilateur
- Compilation en 3 phases :
 - Preprocessing : Suppression des commentaires et traitement des directives de compilation commençant par # ⇒ code source brut
 - Compilation en fichier objet : compilation du source brut ⇒ fichier objet (portant souvent l'extension .obj ou .o sans main) Edition de liens : Combinaison du fichier objet de l'application avec ceux des bibliothèques qu'elle utilise ⇒ fichier exécutable binaire ou une librairie dynamique (.dll sous Windows)
- Compilation ⇒ vérification de la syntaxe mais pas de vérification de la gestion de la mémoire (erreur d'exécution segmentation fault)

Compilation et Exécution



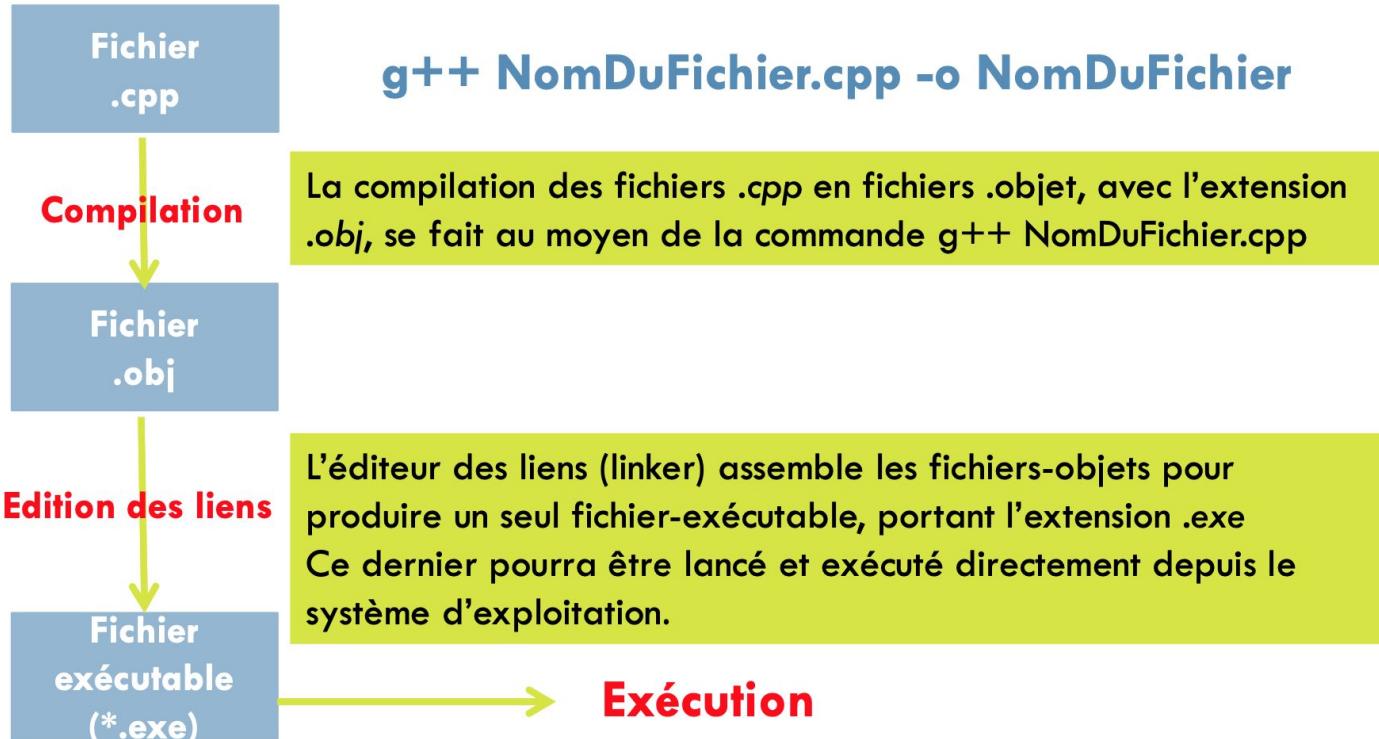
```
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World !" << endl;
    return 0;
}
```

Programme exécutable
par l'ordinateur

Programme en langage C++:
*Fichier texte compréhensible
par un programmeur*

*Fichier compréhensible par
l'ordinateur*

Compilation et Exécution



Erreurs

- Erreurs de compilation: Erreur de syntaxe, déclaration manquante, parenthèse manquante,...
- Erreur de liens: Appel à des fonctions dont les bibliothèques sont manquantes
- Erreur d'exécution: Segmentation fault, overflow, division par zéro
- Erreur logique

Erreurs

Les erreurs détectées **à la compilation** peuvent être classées en plusieurs types :

- **Erreurs de syntaxe**
- **Erreurs sémantiques**
- **Erreurs de portée**

En revanche, certains types d'erreurs **ne sont pas détectés lors de la compilation** :

- **Erreurs d'exécution**
- **Erreurs logiques**

Programmer

Une fois trouvé l'**algorithme**, programmer en C++ comporte **3 phases**:

1. **Editer** le programme – avec votre éditeur de texte favori . . .
2. **Compiler** le programme (avec g++)
3. **Exécuter** le programme
- ...
4. **TESTER et DEBUGGER** : retour au point 1 !

Ça peut durer assez longtemps...

Le langage C++ : Récapitulation

Le langage C++ est un langage compilé :

- le code est figé et ne peut être étendu durant l'exécution
- en contre-partie, gain énorme en performances

Les données peuvent être statiques ou allouées dynamiquement selon les choix du programmeur :

- statiques : gain de performance
- allouées dynamiquement : gain de place mémoire

Le langage C++ permet au programmeur de choisir entre le passage par valeur, par adresse ou par **référence** (plus loin !).

Premier programme C++

Tout ce qui était possible en C, l'est aussi en C++ :

```
/* les commentaires
   à la mode C */

#include <iostream>
#include <cstdio>                      // les librairies C

int main()
{
    std::cout << "Hello" << std::endl;
    printf ("Hello bis\n");           // les instructions C
    return 0;
}
```

En général, on considère que le programme principal « `main` » doit retourner « `0` » s'il s'est correctement exécuté.

Types de base

- ▶ `int` : entiers (au min 16 bits, pour des valeurs ≤ 32767)
(d'autres formats existent: `long int` (min 32 bits), `short int`, `unsigned int`) ...
- ▶ (`float`), `double` et `long double` : nombres à virgule flottante (en général 15 chiffres sign. pour `double`).
Par ex. 23.3 ou 2.456e12 ou 23.56e - 4
- ▶ `char` : caractères ('a','b',... 'A',...,':',...).
- ▶ `bool` : booléens ('true' ou 'false').
(Et: 0 \equiv false ; tout entier non nul équivaut à false)

Définition de variable

Syntaxe : *type v;*

```
int p;  
double x;
```

- ▶ Toute variable doit être définie **avant** d'être utilisée !
- ▶ Une définition peut apparaître **n'importe où** dans un programme.
- ▶ Une variable est définie **jusqu'à la fin de la première instruction composée** (marquée par **}**) qui contient sa définition.
- ▶ (Une variable définie en dehors de toute fonction – et de tout espace de nom – est une **variable globale**).

Définition de variable

Une variable peut être initialisée lors de sa déclaration, deux notations sont possibles :

```
int p=34;  
double x=12.34;
```

```
int p (34);  
double x (12.34);
```

Une variable d'un type élémentaire qui n'est pas initialisée, n'a pas de valeur définie: **elle peut contenir n'importe quoi.**

Types de base

Types de base : char, int, float, double + des modificateurs optionnels :
long, short, unsigned, signed

```
char c, ch;  
c = 'a';  
ch = '\n';  
  
int i, j;  
i = 10;  
  
long int ii;  
long jj;  
ii = 12345;  
jj = 888888L;  
  
int m = 1000;  
long n = 999999L;
```

```
short int mm;  
short nn;  
short pp = 100;  
  
unsigned int il;  
unsigned long jl;  
unsigned short k1;  
il = 345U;  
jl = 888888UL;  
  
signed int i2;  
signed long j2;  
signed short k2;
```

```
float x1, x2;  
float x3 = 0.1F;  
  
double y;  
long double z;  
  
x1 = 0.45;  
x2 = -3.5e12F;  
y = 0.45;  
z = 0.99e-2L;
```

void : pour spécifier qu'il n'y a pas de type

Littéraux caractères

Les littéraux caractères se notent avec des guillemets simples (" quote " en anglais). Eventuellement précédés de " L " pour les caractères longs:

```
char      c1 = 'a';
wchar_t   c2 = L'b';
```

Les caractères inaccessibles au clavier peuvent être fournis grâce à leur code en base 16 (précédé de x) ou en base 8 (simplement précédé de) :

```
char  c1 = 'a';
char  c2 = L'a';
char  c3 = '\141';
char  c4 = '\x61';
```



toutes représentations
du caractère 'a'

Il existe un certain nombre de séquences d'échappement reconnues par le langage C/C++:

| | | | |
|----|----|----|-----|
| \' | \" | \? | \\" |
| \a | \b | \f | \n |
| | | | \r |
| | | | \t |
| | | | \v |

Littéraux entiers

Les littéraux entiers sont des nombres, sans exposant, ni virgule.

- en base décimale (10), les nombres entiers ne peuvent commencer par " 0 "
- en base octale (8), ils doivent commencer par " 0 "
- en base hexadécimale (16), ils doivent commencer par " 0x " ou " 0x "
- les nombres peuvent être précédés d'un opérateur unaire (" - " ou "+")

```
unsigned int a1 = 67;  
unsigned int a2 = 0103;  
unsigned int a3 = 0x43;
```

toutes représentations
du nombre '67'

Littéraux Réels

Les littéraux flottants sont des nombres réels avec un exposant et/ou une virgule. Les nombres peuvent être précédés d'un signe, opérateur unaire (" - " ou " + ") :

```
float x1, x2;  
float x3 = 0.1F;
```

```
double y;  
long double z;
```

```
x1 = 0.45;  
x2 = -3.5e12F;  
y = 0.45;  
z = 0.99e-2L;
```

Rappel : les suffixes " F " et " L ", en minuscules ou majuscules, permettent de préciser respectivement que la constante est simple précision (" float ") ou quadruple précision (" long double "). S'il n'y a aucun suffixe, il s'agit d'une constante double précision (" double ").

Constantes symboliques

Syntaxe : `const type nom = val;`

Par exemple: `const int Taille = 100 ;`

Il ne sera pas possible de modifier Taille dans le reste du programme (erreur à la compilation)...

Littéraux Chaîne de caractère

En C et C++, les littéraux chaînes de caractères sont définis comme des tableaux de n caractères du type " const char ".

```
const char s1[5] = "abcd";    // OK
const char s2[5] = "abcde";   // error: array bounds overflow
s1[0] = 'R';                 // error: l-value specifies
                             // const object

char s3[5] = "abcd";         // OK
s3[0] = 'R';                 // OK
```

En C et C++, les chaînes de caractères se terminent par le caractère zéro (\0) qui compte dans le nombre de caractères. C'est ce que l'on appelle les chaînes de caractères " null terminated " par opposition aux chaînes de la classe string de la librairie standard C++ que nous verrons plus tard.

Littéraux Chaîne de caractère : String

Les chaînes de caractères sont représentées par des tableaux de caractères. Il existe cependant dans la librairie standard de C++ une classe std::string, mais il ne s'agit pas d'un type de base, faisant partie du noyau du langage.

Littéraux Chaîne de caractère : String

Pour l'utiliser, il faut placer tête du fichier :

```
# include <string>
```

- ▶ `string t` ; définit t comme une variable...
- ▶ `string s(25,'a')` ;
- ▶ `string mot = "bonjour"` ;
- ▶ `s.size()` représente la longueur de s
- ▶ `s[i]` est le i-ème caractère de s (`i = 0,1,...s.size()-1`)
- ▶ `s+t` est une nouvelle chaîne correspondant à la concaténation de s et t.

Le type bool

Contrairement à C, le langage C++ dispose d'un type booléen, le type `bool`, dont les valeurs sont `true` et `false`. Les conditions du `if`, `while`, `do`, `for` et de l'expression conditionnelle `? :` sont des expressions de type `bool`. Les opérateurs relationnels et logiques travaillent sur ce nouveau type.

```
int i, j, k;  
bool b;  
  
b = (i>j);    // Bien  
k = (j>i);    // A éviter  
  
while (b)      // Bien  
{  
    // ...  
}  
  
if (k)         // A éviter  
...  
...
```

Pour des raisons de compatibilité avec le C, les conversions implicites entre booléens et types arithmétiques sont conservées.

CONSEIL : les éviter

Tableaux

Pour utiliser la classe `vector`, il faut placer en tête du fichier :

```
# include <vector>
```

Un tableau est typé:

```
vector<int> Tab(100,5) ;  
vector<int> Tab(50) ;  
vector<double> T ;
```

Structure générale: `vector< type > Nom(n,v) ;`

- ▶ `vector< type > Nom1 = Nom2 ;`
→ les valeurs de `Nom2` sont alors recopiées dans `Nom1`.
- ▶ `T.size()` correspond à la taille de `T`.

Les identifiEURS

Les identifiEURS ou symboles permettent de nommer :

- des classes (class), des structures (struct) ou des unions (union)
- des variables, des constantes, ou des instances de classe
- des types énumérés (enum)
- des attributs de classes
- des fonctions ou des méthodes de classes
- des alias de types (typedef)
- des labels (goto), des macros (#define) ou des paramètres de macros

Un symbole C/C++ est formé d'une suite de lettres (a-z A-Z), de chiffres (0-9) ou de soulignés (underscore _). Le premier caractère d'un symbole est obligatoirement une lettre ou un souligné :

```
int    H1B4_8;  
float  __abc_1_2;  
char   x1x2x3;
```

Les identificateurs : Exemple

- ❑ **Sensibilité à la Casse** : Par exemple, "compteur" et "Compteur" sont considérés comme différents.
- ❑ **Règles de Nom** : Les identificateurs doivent commencer par une lettre (majuscule ou minuscule) ou un soulignement (_), suivi de lettres, de chiffres ou de soulignements. Ils ne peuvent pas commencer par un chiffre.
- ❑ **Caractères Spéciaux** : Les caractères spéciaux, tels que les espaces, les signes de ponctuation et les symboles mathématiques, **ne sont pas autorisés dans les identificateurs.**
- ❑ **Mots-Clés Réservés** : Les mots-clés réservés du langage C++ ne peuvent pas être utilisés comme identificateurs.

Les identificateurs : Quiz

Question 1 : Identificateurs Valides

Quels des éléments suivants sont des identificateurs valides en C++ ?

- A) compteur
- B) 123compteur
- C) _salaire_moyen
- D) void

Question 2 : Identificateurs Invalides

Quels des éléments suivants sont des identificateurs invalides en C++ ?

- A) valeur
- B) mon-salaire
- C) NomDeLaClasse
- D) @variable

Les identificateurs : Quiz

Question 1 : Identificateurs Valides

Quels des éléments suivants sont des identificateurs valides en C++ ?

- A) **compteur**
- B) 123compteur
- C) **_salaire_moyen**
- D) void

Question 2 : Identificateurs Invalides

Quels des éléments suivants sont des identificateurs invalides en C++ ?

- A) valeur
- B) **mon-salaire**
- C) NomDeLaClasse
- D) **@variable**

Typologie des langages: Passage des paramètres

Une chose importante à connaître est le mode utilisé par chaque langage pour passer les paramètres aux fonctions et procédures.

- **Passage par valeur** La fonction ne connaît pas la variable de départ, seulement sa valeur. La variable ne peut être modifiée par la fonction/procédure. Comment alors écrire une fonction qui modifie des variables ?
- **Passage par adresse** La variable peut être modifiée par la fonction/procédure. Comment alors distinguer entre celles qui modifient la variable et les autres ?

Passage des paramètres : langage C

Le C utilise aussi le passage par valeur :

```
#include <stdio.h>

void f(double y) {
    y = 2 * y;
}

void g(double *y) {
    *y = 2 * *y;
}

void main()
{
    double x = 5;
    printf("Valeur de x = %g\n", x);

    f(x);
    printf("Valeur de x = %g\n", x);

    g(&x);
    printf("Valeur de x = %g\n", x);
}
```

Passage de paramètres par VALEUR

Par défaut, les paramètres d'une fonction sont initialisés par une copie des valeurs des paramètres réels.

Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres réels.

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;  
b = a ;  
a = aux ; }  
  
...  
  
int main() {  
int x,y ;  
x=5 ;  
y=10 ;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x"   
b = a ;  
a = aux ; }             "y"   
...  
  
int main() {  
    int x,y;  
    x=5 ;  
    y=10 ;  
    permut(y,x) ;  
    cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x"   
b = a ;                  "y"   
a = aux ; }  
...  
  
int main() {  
int x,y;  
x=5;  
y=10;  
permut(y,x);  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;  
a = aux ; }             "y" 10  
...  
  
...
```

```
int main() {  
int x,y;  
x=5;  
y=10;  
permut(y,x);  
cout<<" x = "<<x; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;  
a = aux ; }             "y" 10  
...
```

```
int main() {  
int x,y;  
x=5;  
y=10;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;                 "y" 10    "a" 10  
a = aux ; }             ...      "b" 5  
  
int main() {  
int x,y;  
x=5;  
y=10;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;                 "y" 10   "a" 10  
a = aux ; }             ...      "b" 5  
  
int main() {           "aux" 5  
int x,y ;  
x=5 ;  
y=10 ;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;  
a = aux ; }             "y" 10   "a" 10  
...                      "b" 10  
  
int main() {  
int x,y ;  
x=5 ;  
y=10 ;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;                 "y" 10   "a" 5  
a = aux ; }             ...      "b" 10  
  
int main() {             "aux" 5  
int x,y ;  
x=5 ;  
y=10 ;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;           "x" 5  
b = a ;  
a = aux ; }             "y" 10  
...
```

```
int main() {  
int x,y;  
x=5;  
y=10;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage de paramètres par VALEUR

```
void permut(int  
a,int b)  
{int aux = b ;  
b = a ;  
a = aux ; }  
...
```

“x”
“y”

```
int main() {  
int x,y ;  
x=5 ;  
y=10 ;  
permut(y,x) ;  
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

Pour modifier la valeur d'un paramètre réel dans une fonction, il faut passer ce paramètre par **référence**.

Une référence sur une variable est un **synonyme** de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

On utilise le symbole **&** pour la déclaration d'une référence:

Dans la liste des paramètres de la définition d'une fonction, **type & *p_i*** déclare le paramètre ***p_i*** comme étant une référence sur le *i^{eme}* paramètre réel ***v_i***.

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a;
a = aux ; }

int main() {
int x,y;
x=5;
y=10;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ; }
...
```

“x”

“y”

```
int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ; }
...
```

“x”
“y”

```
int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b;
 b = a;
 a = aux; }
...
```

“x” 5
“y” 10

```
int main() {
int x,y;
x=5;
y=10;
permut(y,x);
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ; }
...
```

“x” 5
“y” 10

```
int main() {
int x,y;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b;
 b = a;
 a = aux; }
...
```

“b” “x” 5
“a” “y” 10

```
int main() {
int x,y;
x=5;
y=10;
permut(y,x);
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a;
a = aux; }
...
```

“b” “x” 5
“a” “y” 10

```
int main() {
int x,y;
x=5;
y=10;
permut(y,x);
cout<<" x = "<<x; }
```

“ aux ” 5

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b;
b = a;
a = aux; }
...
```

“b” “x” 10
“a” “y” 10

```
int main() {
int x,y;
x=5;
y=10;
permut(y,x);
cout<<" x = "<<x; }
```

“ aux ” 5

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b;
b = a;
a = aux; }
...
```

“b” “x” 10

“a” “y” 5

```
int main() {
int x,y;
x=5;
y=10;
permut(y,x);
cout<<" x = "<<x; }
```

“ aux ” 5

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ; }
...
```

“x” 10
“y” 5

```
int main() {
int x,y ;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

Passage des paramètres par RÉFÉRENCE

```
void permut(int & a,int & b)
{int aux = b ;
b = a ;
a = aux ; }
...
```

“x” 10

“y” 5

```
int main() {
int x,y ;
x=5 ;
y=10 ;
permut(y,x) ;
cout<<" x = "<<x ; }
```

x = 10

Passage des paramètres : langage C

Le C utilise aussi le passage par valeur :

```
#include <stdio.h>

void f(double y) {
    y = 2 * y;
}

void g(double *y) {
    *y = 2 * *y;
}

void main()
{
    double x = 5;
    printf("Valeur de x = %g\n", x);

    f(x);
    printf("Valeur de x = %g\n", x);

    g(&x);
    printf("Valeur de x = %g\n", x);
}
```

Passage des paramètres : langage C

Le C utilise aussi le passage par valeur :

```
#include <stdio.h>

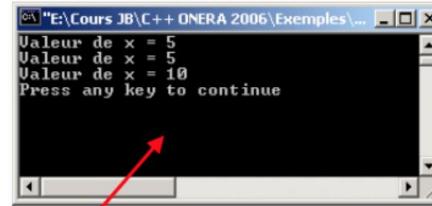
void f(double y) {
    y = 2 * y;
}

void g(double *y) {
    *y = 2 * *y;
}

void main()
{
    double x = 5;
    printf("Valeur de x = %g\n", x);

    f(x);
    printf("Valeur de x = %g\n", x);

    g(&x);
    printf("Valeur de x = %g\n", x);
}
```



Passage des paramètres : Récapitulation

Une chose importante à connaître est le mode utilisé par chaque langage pour passer les paramètres aux fonctions et procédures.

- **Passage par valeur** La fonction ne connaît pas la variable de départ, seulement sa valeur. La variable ne peut être modifiée par la fonction/procédure. Comment alors écrire une fonction qui modifie des variables ?
- **Passage par adresse** La variable peut être modifiée par la fonction/procédure. Comment alors distinguer entre celles qui modifient la variable et les autres ?

Typologie des langages: passage des paramètres

| | Passage par valeur | Passage par référence |
|---|---|---|
| MATLAB | OUI | Le langage l'utilise "dans le dos" du programmeur |
| Fortran, Lisp, Smalltalk, Python | NON | OUI |
| C | OUI => on passe les adresses des variables pour les modifier | NON |
| C++ | oui, par défaut | possible, au choix du programmeur |
| Java | oui, types simples "dans le dos" du programmeur | oui, objets "dans le dos" du programmeur |

Cin et Cout

Entrées/sorties fournies à travers la librairie iostream

cout << expr1 << ... << exprn

- Instruction affichant expr1 puis expr2, etc.
- cout : flux de sortie associé à la sortie standard (stdout)
- << : opérateur binaire associatif à gauche, de première opérande cout et de 2ème l'expression à afficher, et de résultat le flot de sortie
- << : opérateur surchargé (ou sur-défini). Utilisé aussi bien pour les chaînes de caractères, que les entiers, les réels etc.

cin >> var1 >> ... >> varn

- Instruction affectant aux variables var1, var2, etc. les valeurs lues (au clavier)
- cin : flux d'entrée associée à l'entrée standard (stdin)
- >> : opérateur similaire à <<

Cout

cout représente la fenêtre Terminal

affiche la *valeur* de la variable n
(et non pas la lettre n)

```
cout << "La variable n contient "
```

ce qui est entre guillemets ("") est affiché littéralement

les différents éléments à afficher doivent être séparés par le symbole <<

affiche:

La variable n contient 4.

fait un "retour à la ligne": le prochain affichage se fera sur la ligne suivante de la fenêtre Terminal

Cout

- Le flot **cout** est un flot de sortie prédéfini, connecté à la sortie standard stdout (l'écran).
- L'opérateur `<<` permet d'envoyer de l'information sur le flot **cout**, correspondant à l'écran.
- En générale, l'opérateur `<<` permet d'envoyer sur le flot **cout** la valeur d'une expression d'un type de base quelconque.
- On peut aussi utiliser cout pour afficher la valeur d'une **expression**:

```
cout << "Le carré de " << n << " est " << n * n << "." << endl;
```

Cout

Exemple 1:

Considérons ces instructions :

```
int n = 20 ;
```

```
cout << "Valeur : " ;
```

```
cout << n ;
```

Elles affichent le résultat suivant :

```
Valeur : 20
```

Cout

Exemple 1: (suite)

Les deux instructions :

```
cout << "Valeur : " ;
```

```
cout << n ;
```

peuvent se condenser en une seule :

```
cout << "Valeur : " << n ;
```

Cout

Remarque:

- `cout` et `endl` sont des mots réservé de la bibliothèque standard `std` (using namespace `std`), nous pouvons les nommer `std::cout` et `std::endl`.

```
#include <iostream>
int main(){
    std::cout << "Hello World !" << std::endl;
    return 0;
}
```

Cout

Exemple :

```
int n(4);
int n_carre;

n_carre = n * n;

cout << "La variable n contient " << n << "." << endl;
cout << "Le carre de " << n << " est " << n_carre << "." << endl;
cout << "Le double de n est " << 2 * n << "." << endl;
```

Output :

Cout

Exemple :

```
int n(4);
int n_carre;

n_carre = n * n;

cout << "La variable n contient " << n << "." << endl;
cout << "Le carre de " << n << " est " << n_carre << "." << endl;
cout << "Le double de n est " << 2 * n << "." << endl;
```

Output :

La variable n contient 4.
Le carre de 4 est 16.
Le double de n est 8.

Cin

cin représente le clavier

les différents éléments sont séparés par le symbole >>

nom de la variable dans laquelle sera stockée la valeur entrée au clavier

```
cin >> n;
```

- **Attention**, uniquement des noms de variables peuvent figurer à droite du symbole >>.
- Le **flot cin** est un flot d'entrée prédéfini, connecté à l'entrée standard stdin (le clavier).

Cin

- On ne peut pas faire :
cin >> "Entrez un nombre" >> n;

Cin

- On ne peut pas faire :

cin >> "Entrez un nombre" >> n;

Il faut faire:

cout << "Entrez un nombre" << endl;

cin >> n;

Cin

cin >> n1 >> n2 >> n3;

Exemple:

```
int main() {
    int n1 = 0, n2 = 0;
    cout << "Saisir deux entiers" << endl;
    cin >> n1 >> n2 ;
    cout << n1 << ", " << n2<< endl;
}
```

Cin

```
int n;  
  
cout << "Entrez une valeur pour n:";  
cin >> n;  
  
cout << "La variable n contient " << n << "." << endl;
```

Cin

```
int n;  
  
cout << "Entrez une valeur pour n:";  
cin >> n;  
  
cout << "La variable n contient " << n << "." << endl;
```

Output:

Entrez une valeur pour n:

2

La variable n contient 2.

Cin et Cout : Quiz

Question 1 : Quelles sont les fonctions en C++ utilisées pour la saisie et la sortie de données standard ?

- A) scan et print
- B) input et output
- C) cin et cout
- D) read et write

Question 2 : Quelle est la fonction utilisée pour lire des données depuis la console en C++ ?

- A) read
- B) cin
- C) output
- D) print

Cin et Cout : Quiz

Question 3 :

Comment affichez-vous le contenu de la variable nom à l'aide de cout en C++ et saisissez une valeur pour la variable prenom à l'aide de cin ?

- A) cin >> prenom; cout << nom;
- B) cout << cin >> nom; cin << prenom;
- C) cout >> nom; cin << prenom;
- D) cin << nom; cout >> prenom;

Exercice 1 : Calcul de la somme

Écrivez un programme en C++ qui demande à l'utilisateur de saisir deux nombres, puis affiche la somme de ces deux nombres.

Exercice 2 : Calcul de l'aire d'un rectangle

Écrivez un programme en C++ qui demande à l'utilisateur de saisir la longueur et la largeur d'un rectangle, puis calcule et affiche son aire.

Cin et Cout

Possibilité de modifier la façon dont les éléments sont lus ou écrits dans le flux :

| | |
|----------------------------------|--|
| <code>dec</code> | lecture/écriture d'un entier en décimal |
| <code>oct</code> | lecture/écriture d'un entier en octal |
| <code>hex</code> | lecture/écriture d'un entier en hexadécimal |
| <code>endl</code> | insère un saut de ligne et vide les tampons |
| <code>setw(int n)</code> | affichage de <i>n</i> caractères |
| <code>setprecision(int n)</code> | affichage de la valeur avec <i>n</i> chiffres avec éventuellement un arrondi de la valeur |
| <code>setfill(char)</code> | définit le caractère de remplissage |
| <code>flush</code> | vide les tampons après écriture |

```
#include <iostream.h>
#include <iomanip.h> // attention à bien inclure cette librairie

int main() {
    int i=1234;
    float p=12.3456;
    cout << "|" << setw(8) << setfill('*')
        << hex << i << "|" << endl << "|"
        << setw(6) << setprecision(4)
        << p << "|" << endl;
}
```

```
| *****4d2 |
| *12.35 |
```

Affectation

En C/C++, l'affectation est une expression:

Soient v une variable (au sens large) et $expr$ une expression.

$v = expr$ affecte la valeur de $expr$ à la variable v et retourne la valeur affectée à v comme résultat.

Par exemple, $i = (j = 0)$ affecte 0 à j puis à i et retourne 0 !!

Les opérateurs et les expressions

- Les **opérateurs** sont des éléments syntaxiques qui effectuent certaines opérations en utilisant les valeurs de leurs **opérandes** (paramètres de l'opération).



- #### ▪ Opérateurs unaires (monadiques) 1 opérande

-b Opérateur monadique moins

`a++` Opérateur monadique de post-incrémentation

- ### ▪ Opérateurs binaires (dyadiques) *2 opérandes*

a * b L'opérateur de multiplication est binaire

`y = z` l'affectation est également binaire

(int)y Transtypage (casting)

- Un seul opérateur ternaire

`x > y ? x : y` Retourne la valeur maximale entre x et y

Pré et Post incrément

- ▶ ++var incrémentera la variable var et retourne la nouvelle valeur.
(++i équivaut à $\text{i}=\text{i}+1$)
- ▶ $\text{var}++$ incrémentera la variable var et retourne l'ancienne valeur.
($\text{i}++$ équivaut à $(\text{i}=\text{i}+1)-1$)

En dehors d'une expression, $\text{i}++$ et ++i sont donc équivalentes.

Pré et Post incrément

- Si l'on prend l'exemple suivant :

```
int i(3);  
int j(i);      // i et j ont la même valeur  
int k(0);  
int l(0);  
k = ++i;      // opérateur préfixé  
l = j++;      // opérateur postfixé
```

Pré et Post incrément

- Si l'on prend l'exemple suivant :

```
int i(3);  
int j(i);      // i et j ont la même valeur  
int k(0);  
int l(0);  
k = ++i;      // opérateur préfixé  
l = j++;      // opérateur postfixé
```

- A l'issue de ce bout de code:

i et j auront tous les deux la valeur 4, mais k aura la valeur 4 alors que l aura la valeur 3.

Pré et Post incrément et décrément

| | | |
|--------------------------------|--------------|--------------------------------------|
| <pre>int i = 0; i++;</pre> | | <pre>int i = 0; i = i + 1;</pre> |
| <pre>int t = i++;</pre> | | <pre>int t = i; i = i + 1;</pre> |
| <pre>int s = ++i;</pre> | équivalent à | <pre>i = i + 1; int s = i;</pre> |
| <pre>int r = --i;</pre> | | <pre>i = i - 1; int r = i;</pre> |
| <pre>r += i;</pre> | | <pre>r = r + i;</pre> |
| <pre>s *= i; i /= 2;</pre> | | <pre>s = s * i; i = i / 2;</pre> |

Opérateurs classiques

- ▶ Opérateurs arithmétiques:
*, +, -, / (division entière et réelle), % (modulo)
- ▶ Opérateurs de comparaison
< (inférieur), <= (inférieur ou égal), == (égal), > (supérieur),
>= (supérieur ou égal) et != (différent)
- ▶ Opérateurs booléens
`&&` représente l'opérateur “ET”, `||` représente le “OU”, et `!` représente le “NON”.
Par exemple, `((x<12) && ((y>0) || !(z>4)))`

Les opérateurs arithmétiques

L'ordre des opérateurs suit les standards mathématiques

1. Parenthèses ()
2. Opérateurs unaires + et - ont la priorité la plus élevée
3. Multiplication (*), division (/) et modulo (%)
4. Addition (+) et soustraction (-)

Exemple:

| | | |
|-----------------------|------------|-----------------------------|
| ■ $x + y * z$ | équivaut à | $x + (y * z)$ |
| ■ $x * y + z \% t$ | équivaut à | $(x * y) + (z \% t)$ |
| ■ $- x \% y$ | équivaut à | $(- x) \% y$ |
| ■ $- x + y \% z$ | équivaut à | $(- x) + (y \% z)$ |
| ■ $- x / - y + z$ | équivaut à | $((- x) / (- y)) + z$ |
| ■ $- x / - (y + z)$ | équivaut à | $(- x) / (- (y + z))$ |

Evaluation des expressions arithmétiques

Si une (sous-)expression mélange plusieurs types, c'est le type le plus large qui est utilisé.

```
int i=3,j=2,m;  
double r=3.4;  
m = (i/j)*r;
```

Evaluation des expressions arithmétiques

Si une (sous-)expression mélange plusieurs types, c'est le type le plus large qui est utilisé.

```
int i=3,j=2,m;  
double r=3.4;  
m = (i/j)*r;
```

- ▶ D'abord l'expression `(i/j)` est évaluée: `/` désigne **ici** la division entière, cela donne donc 1.

Evaluation des expressions arithmétiques

Si une (sous-)expression mélange plusieurs types, c'est le type le plus large qui est utilisé.

```
int i=3,j=2,m;  
double r=3.4;  
m = (i/j)*r;
```

- ▶ D'abord l'expression `(i/j)` est évaluée: `/` désigne **ici** la division entière, cela donne donc 1.
- ▶ Pour évaluer le produit `1*r`, il faut convertir 1 en `double` (1.0) et faire le produit sur les doubles, cela donne 3.4

Evaluation des expressions arithmétiques

Si une (sous-)expression mélange plusieurs types, c'est le type le plus large qui est utilisé.

```
int i=3, j=2, m ;  
double r=3.4 ;  
m = (i/j)*r ;
```

- ▶ D'abord l'expression `(i/j)` est évaluée: `/` désigne **ici** la division entière, cela donne donc 1.
- ▶ Pour évaluer le produit `1*r`, il faut convertir 1 en `double` (1.0) et faire le produit sur les doubles, cela donne 3.4
- ▶ Pour l'affectation, comme `m` est entier, 3.4 est converti en `int`. Finalement on a `m = 3`.

Evaluation des expressions arithmétiques

- Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m ;  
double r=3.4 ;  
m = (double(i)/j)*r ;
```

Donne...

Evaluation des expressions arithmétiques

- Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m ;  
double r=3.4 ;  
m = (double(i)/j)*r ;
```

Donne... 5 !

Evaluation des expressions arithmétiques

- Pour éviter les erreurs, il est possible de convertir explicitement des données d'un certain type en un autre.

Par exemple:

```
int i=3,j=2,m ;  
double r=3.4 ;  
m = (double(i)/j)*r ;
```

Donne... 5 !

- L'évaluation d'une expression arithmétique ne se fait pas toujours de gauche à droite !

Ex: $(i/j)*(r/3)$

Les opérateurs logiques

- **&&** : ET logique (**and**)
- **||**: OU logique (**or**)
- **!:** Négation (**not**)

Evaluation des expressions booléennes

```
if (i >=0 && T[i] > 20) blabla
```

```
if (i<0 || T[i] > 20) blabla
```

Evaluation des expressions booléennes

- ▶ Dans `e1 && e2`, la sous-expression `e2` n'est évaluée que si `e1` a été évaluée à 'true'.

```
if (i >=0 && T[i] > 20) blabla
```

- ▶ Dans `e1 || e2`, la sous-expression `e2` n'est évaluée que si `e1` a été évaluée à 'false'.

```
if (i<0 || T[i] > 20) blabla
```

Evaluation des expressions booléennes

Rappel:

- Pour le ET logique (and):
les deux conditions doivent être vraies;
- Pour le OU logique (or):
au moins l'une des conditions doit être vraie.

L'opérateur à trois opérandes:

condition ? vrai : faux

Exemple:

$y < 5 ? 4 * y : 2 * y$

L'opérateur à trois opérandes:

condition ? vrai : faux

Exemple:

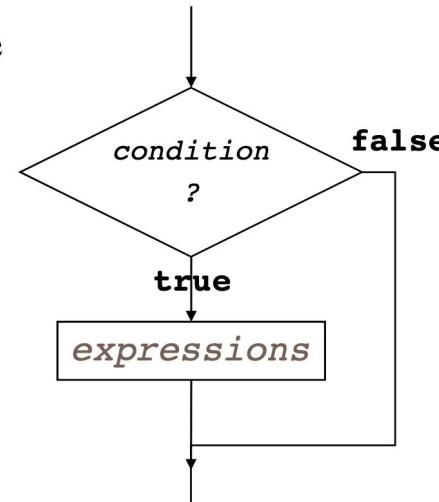
$y < 5 ? 4 * y : 2 * y$

Si *condition* est vrai, alors on retourne l'évaluation de l'expression *vrai*, sinon on retourne celle de *faux*.

Les instructions de contrôle en C++

Décisions

- Interruption de la progression linéaire du programme suite à une décision
 - Basée sur une **condition** qui peut être
 - true
 - false



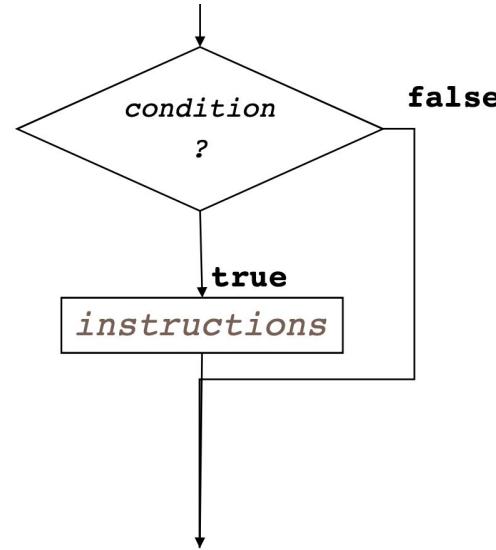
Les instructions de contrôle en C++

Décisions

- Conditions basées sur des opérateurs relationnels
 - e.g. « égal à », « supérieur à »...
- Opérateurs relationnels combinées avec opérations logiques
 - e.g. « ET », « OU »...
- Traduction en code par des structures:
 - **if**
 - **while**
 - **for**
 - **do-while**

L'instruction conditionnelle : if

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



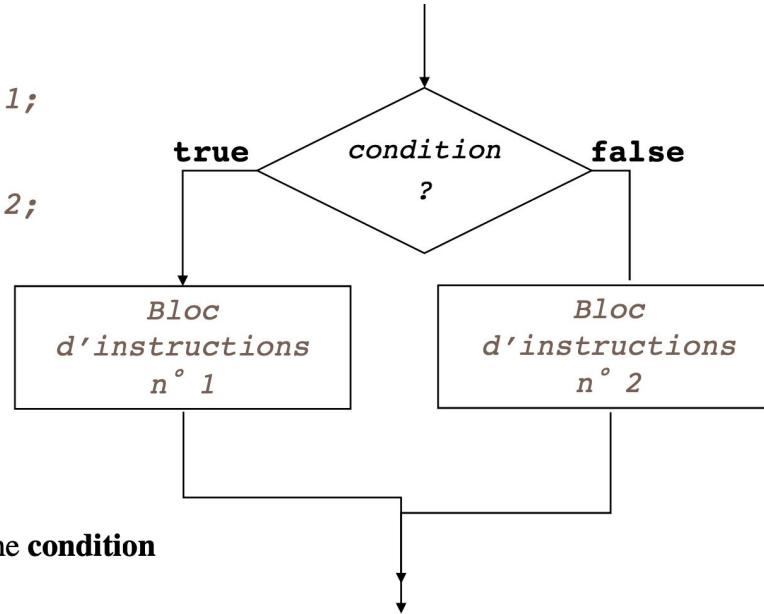
- L'instruction **if** fait apparaître une **condition** entre parenthèses.
- **Attention:** la condition est toujours entourée de parenthèses.

L'instruction conditionnelle : if

```
if (age >= 18) status = "Adult";  
  
if ((age<16) || isStudent) {  
    canHaveAllocations = true;  
    amount = 400;  
}
```

L'instruction conditionnelle : if ... else

```
if (condition) {  
    Bloc d'instructions n°1;  
} else {  
    Bloc d'instructions n°2;  
}
```

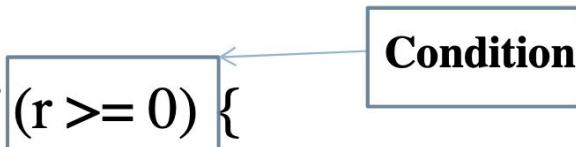


- L'instruction **if** fait apparaître une **condition** entre parenthèses.
- **Attention:** la condition est toujours entourée de parenthèses.

L'instruction conditionnelle : if ... else

```
if (r >= 0) {  
    carre = sqrt(r);      //Cette instruction sera exécutée si la condition est vraie.  
} else {  
    cout<<"Erreur";    //Cette instruction sera exécutée si la condition est fausse.  
}
```

Condition



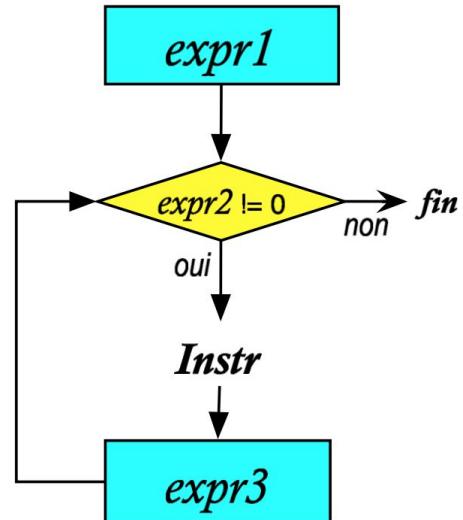
L'instruction conditionnelle : if ... else

```
if (average >= 10)
    cout << "Pass"; //Cette instruction sera exécutée si la condition est vraie.
else {
    cout << "Fail";
    repeat = true;
}
```

N. B: Quand un bloc contient une seule instruction, il n'est pas obligatoire d'utiliser des accolades.

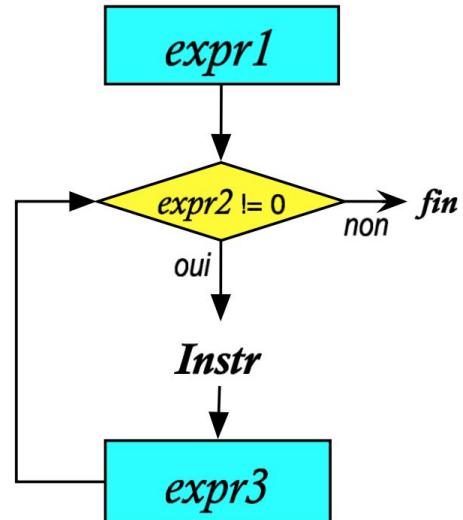
La boucle : FOR

```
for (expr1 ; expr2 ; expr3) instr
```



La boucle FOR

```
for (expr1 ; expr2 ; expr3) instr
```

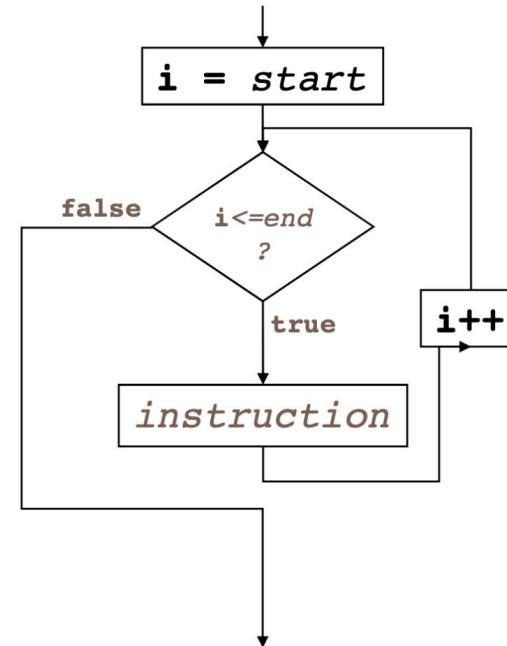


Ex: `for(i=0 ;i<235 ;i=i+1) cout << T[i] ;`
`for(i=0,j=1 ;i<235 ;i=i+1,j=j+3) cout << T[i][j] ;`

La boucle FOR

Syntaxe de l'instruction for

```
for(déclaration et initialisation; condition; incrémentation)  
{  
    bloc d'instructions;  
}
```



La boucle FOR

Remarque:

- L'initialisation, la condition, et l'incrémentation sont séparées par des points-virgules.
- Si la condition ne devient jamais fausse, les instructions dans la boucle sont répétées indéfiniment !
- Comme pour le **if**, les accolades ne sont obligatoires que si plusieurs instructions doivent être répétées.

La boucle FOR

L'instruction for

- La première expression correspond à l'initialisation d'un compteur;
 - Elle est évaluée (une seule fois) avant d'entrer dans la boucle.
- La deuxième expression correspond à la condition de poursuite:
 - Elle conditionne la poursuite de la boucle. Elle est évaluée avant chaque parcours.
- La troisième expression correspond à l'incrémentation du compteur.
 - Elle est évaluée à la fin de chaque parcours.
- Lorsque la condition (l'expression booléenne) est absente, elle est considérée comme vraie.

La boucle FOR

□ Exemple 1:

//Affiche les carrés des 5 premiers entiers.

```
for ( int i(0) ; i < 5 ; ++i ){
    cout << "Le carre de " << i << " vaut " << i * i << endl;
}
```

Le carre de 0 vaut 0

Le carre de 1 vaut 1

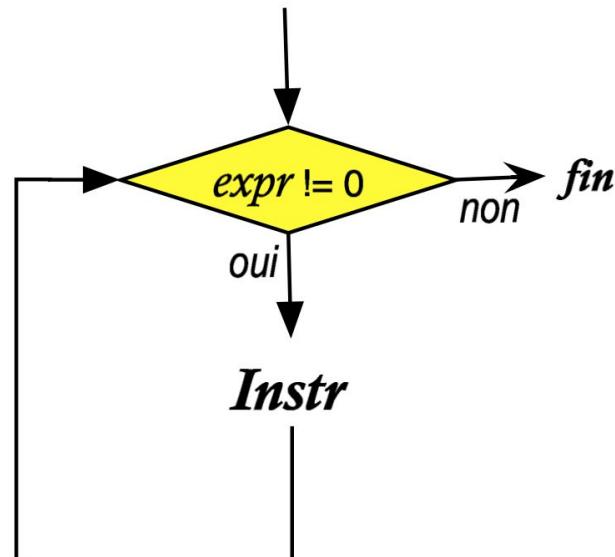
Le carre de 2 vaut 4

Le carre de 3 vaut 9

Le carre de 4 vaut 16

La boucle : WHILE

`while (expr) instr`

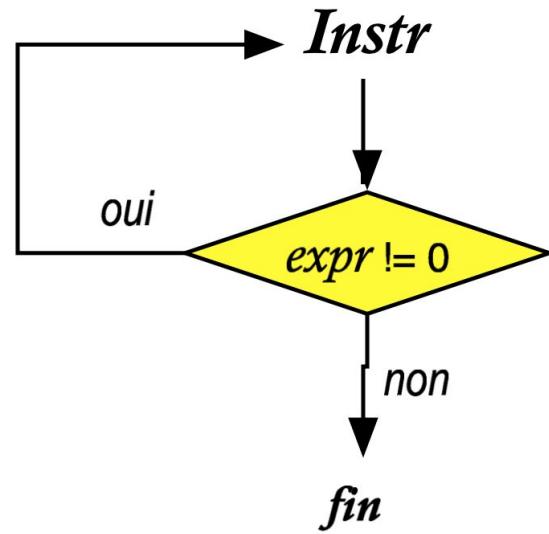


La boucle : WHILE

```
i = valeurInitiale; // Initialisation
while (i <= valeurFinale)
{
    bloc d'instructions;
    i++; //Incrémantation
}
```

La boucle : DO

```
do instr while (expr) ;
```



La boucle : DO

```
int nbre ;  
do{  
    cout << "Donnez un nombre positif (nbre >0) : " << endl;  
    cin >> nbre ;  
    cout << "Vous avez fourni : " << nbre << "\n" ;  
}while (nbre <= 0) ;  
cout << "Réponse correcte" ;
```

Comparaison entre while et do...while

```
int i(10);  
do {  
cout << "Hi" << endl;  
} while (i < 1);  
affichera une fois Hi.
```

```
int i(10);  
while (i < 1) {  
cout << "Hi" << endl;  
}  
n'affichera rien.
```

Dans les 2 cas: la condition $i < 1$ est fausse.

Erreurs classiques

Remarque:

- Il n'y a pas de ; à la fin du **while**...:

while (i < 1); // !!

++i;

sera interprété comme

while(i < 1)

;

++i;

- En revanche, il y a un point-virgule à la fin du do..while:

do {

++i;

} **while**(i < 1);

Les boucles en C++

Choisir la boucle while/la boucle do-while/la boucle for?

- Quand le nombre d'itérations (de répétitions) est connu avant d'entrer dans la boucle, utiliser **for**:

```
for ( int i (0); i < nombre_d_iterations; ++i) {}
```

- Sinon, utiliser **while**:

- Quand les instructions doivent être effectuées au moins une fois, utiliser **do...while**:

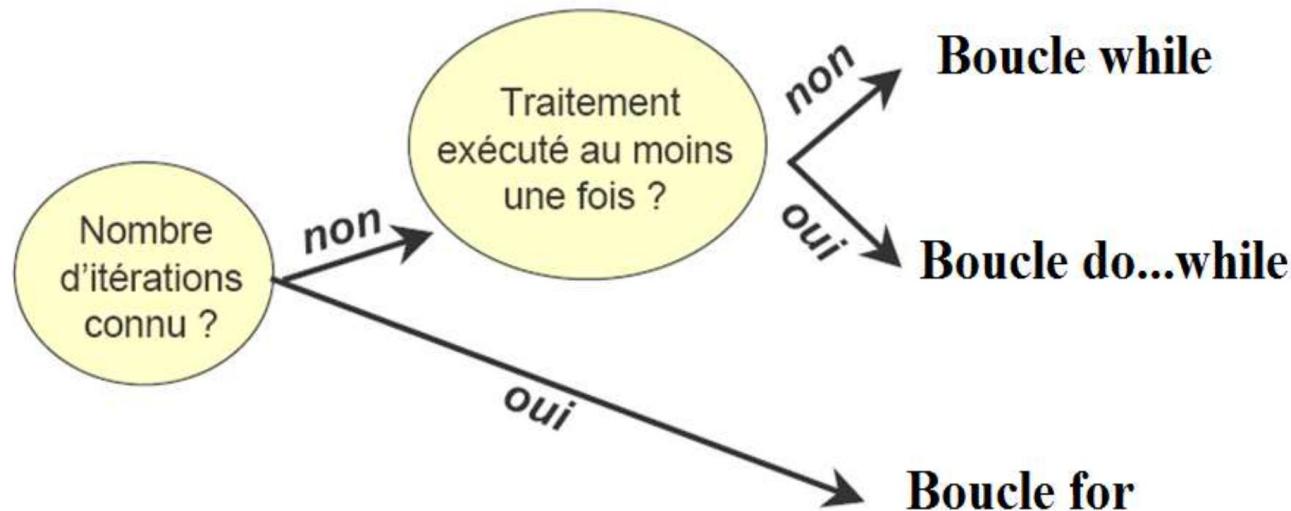
```
do {  
    instructions;  
} while (condition);
```

- Sinon, utiliser la forme **while...**

```
while (condition) {  
    instructions;  
}
```

Les boucles en C++

Choisir la boucle while/la boucle do-while/la boucle for?



Les boucles en C++ : Quiz

Que s'affiche-t-il quand on exécute le code :

```
for(int i(0); i < 3; ++i) {  
    for(int j(0); j < 4; ++j) {  
        if (i == j) {  
            cout << "*";  
        } else {  
            cout << j;  
        }  
    }  
    cout << endl;  
}
```

A:

*123

*123

*123

C:

B:

012*

012*

012*

D:

*123

0*23

01*3

?

Les boucles en C++ : Quiz

Que s'affiche-t-il quand on exécute le code :

```
for(int i(0); i < 3; ++i) {  
    for(int j(0); j < i; ++j) {  
        cout << j;  
    }  
    cout << endl;  
}
```

A:

0

01

B:

0

01

012

C:

rien

D:

0123

0123

0123

?

Définition de fonction

```
type nom( liste des paramètres) { corps }
```

- ▶ *type* est le type du résultat de la fonction.
(*void* si il s'agit d'une procédure)
- ▶ La liste des paramètres (*paramètres formels*):
type₁ *p₁*, ..., *type_n* *p_n*
- ▶ Le *corps* décrit les instructions à effectuer.
Le corps utilise ses propres variables locales, les éventuelles variables globales et les paramètres formels.
- ▶ Si une fonction renvoie un résultat, il doit y avoir (au moins) une instruction *return expr ;*

Définition de fonction : Exemple

```
int max(int a,int b)
{
    int res=b ;
    if (a>b) res = a ;
    return res ;
}
```

Appel d'une fonction

`nom(liste des arguments)`

La liste des arguments (*paramètres réels*) est $expr_1, expr_2, \dots, expr_n$ où chaque $expr_i$ est compatible avec le type $type_i$ du paramètre formel `pi`.

Exemple :

```
int k=34, t=5, m;  
m = max(k,2*t+5);
```

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;
return res ; }

...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;           "x" 
return res ; }                 "y" 
...
```

```
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;           "x" 
return res ; }                 "y" 
...                            
```

```
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;           "x" 5
return res ; }                "y" 10
...

```

```
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a;
return res; }
...
int main() {
int x,y;
x=5;
y=10;
int z = max(y,x);
cout<<" z = "<<z ; }
```

“x”

“y”

“z”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;
 return res ; }
...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

“x”

“y” “a”

“z” “b”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;
 return res ; }
...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

“x”

“y” “a”

“z” “b”

“res”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;
 return res ; }
...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

“x”
“y” “a”
“z”
“b”
“res”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b;
 if (a>b) res = a;
 return res; }
...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

“x”

“y” “a”

“z” “b”

“res”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
 if (a>b) res = a ;
 return res ; }
...
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

“x”

“y”

“z”

Définition de fonction : Exemple

```
int max(int a,int b)
{int res=b ;
if (a>b) res = a ;
return res ; }
```

...

```
int main() {
int x,y ;
x=5 ;
y=10 ;
int z = max(y,x) ;
cout<<" z = "<<z ; }
```

► Fin

Namespaces

- Utilisation d'espaces de noms (namespace) lors de l'utilisation de nombreuses bibliothèques pour éviter les conflits de noms
- Espace de noms : association d'un nom à un ensemble de variable, types ou fonctions Ex. Si la fonction MaFonction() est définie dans l'espace de noms MonEspace, l'appel de la fonction se fait par MonEspace::MaFonction()
- Pour être parfaitement correct : std::cin std::cout std::endl
- Pour éviter l'appel explicite à un espace de noms : using
using std::cout ; // pour une fonction spécifique
using namespace std; // pour toutes les fonctions

Namespaces

- ❑ En C++, un **namespace** (espace de noms) est une fonctionnalité qui permet de regrouper un ensemble de noms de variables, de fonctions et de classes sous **un nom unique** pour éviter les conflits de noms dans un programme.
- ❑ Les **namespaces** sont utilisés pour **organiser et hiérarchiser le code**, ce qui est particulièrement utile dans les projets de grande envergure ou lorsque des bibliothèques tierces sont utilisées.

Déclaration d'un namespace :

```
namespace MonNamespace {  
    // Déclarations de variables, fonctions, classes, etc.  
    int variable1;  
    void fonction1();  
    class MaClasse;  
}  
  
int main() {  
    MonNamespace::variable1 = 42;  
    MonNamespace::fonction1();  
    MonNamespace::MaClasse objet;  
    return 0;  
}
```

Déclaration, règles d'identification et portée des variables

- Toute variable doit être déclarée avant d'être utilisée
- Constante symbolique : const int taille = 1000; // Impossible de modifier taille dans la suite du programme
- La portée (visibilité) d'une variable commence à la fin de sa déclaration jusqu'à la fin du bloc de sa déclaration

```
// une fonction nommée f de paramètre i
void f (int i) {  int j; // variable locale
                  j=i;
}
```

- Toute double déclaration de variable est interdite dans le même bloc

```
int i,j,m; // variable globale
void f(int i) {
              int j,k; // variable locale
              char k; // erreur de compilation
              j=i;
              m=0;
}
```

Fin de partie

Introduction au langage C++