Chapitre II : Principes de base du langage C++

II.1 Introduction

Le langage C++ peut être considéré comme un perfectionnement du langage C qui offre les possibilités de la POO.
Les notions de base de la programmation en C restent valables en C++, néanmoins C et C++ diffèrent sur quelques conventions (déclaration des variables et des fonctions, nouveaux mots clés…)
Ce chapitre retrace ses différences, et traite les autres outils de la programmation structurée ajouté à C++

II.2 Structure générale d'un programme

II.2.1 Fonction main

Tout programme doit avoir un point d'entrée nommé ***main***

```
int main()
{
   return 0;
}
```

La fonction ***main*** est la fonction appelée par le système d'exploitation lors de l'exécution du programme

- { et } délimitent le corps de la fonction
- ***main*** retourne un entier au système: 0 (zéro) veut dire succès
- Chaque expression doit finir par; (point virgule)

II.2.2 Commentaires

En C et C++: Commentaires sur plusieurs lignes : délimités par **/*** (début) et ***/** (fin).

```
 /* Un commentaire en une seule ligne */
/*
* Un commentaire sur plusieurs
* lignes
*/
```

En C++ uniquement : Commentaires sur une seule ligne : délimités par **//** (début) et fin de ligne (n'existe pas en C)

```
    // Un commentaire jusqu'à la fin de cette ligne
```

II.2.3 Fichiers Sources

Un programme est généralement constitué de plusieurs modules, chaque module est composé de deux fichiers sources:

- Un fichier contenant la description de l'interface du module
- Un fichier contenant l'implémentation proprement dite du module

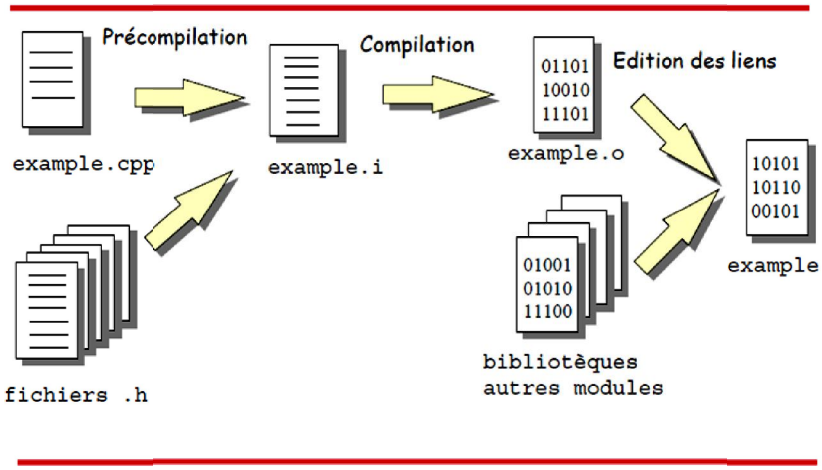
Un suffixe est utilisé pour déterminer le type de fichier

- .h, .H, .hpp, .hxx : pour les fichiers de description d'interface (header files ou include files)
- .c, .cc, .cxx, .cpp, .c++ : pour les fichiers d'implémentation

Dans un fichier source on peut trouver:

- Commentaires
- Instructions pré-processeur
- Instructions C++

II.2.4 Construction de l'Exécutable [1]



II.2.5 Bibliothèque de C++

C++ possède une bibliothèque très riche, qui comporte un très grand nombre d'outils (fonctions, types, …) qui permettent de faciliter la programmation. Elle intègre en plus de la bibliothèque standard de C, des librairies de gestion des entrées-sorties ainsi que des outils de manipulation des chaînes de caractères, des tableaux et d'autres structures de données.
Pour utiliser, en C++, les outils qui existaient dans la bibliothèque standard de C (stdio.h, string.h, …) ainsi que certains nouveaux outils, il suffit de spécifier avec la directive *include* le fichier entête (.h) souhaité.

```
#include <iostream>  // En C++ : « Input Output Stream » ou bien « Flux d'entrée-sortie »
#include <cstdio>    // En C : « Flux d'entrée-sortie »
```

Exemple:

```
#include <iostream>
#include <cstdio>     // Les librairie C

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

II.2.6 Espace de noms en C++

Pour des raisons liées à la POO (généricité, modularité…) et pour éviter certains conflits qui peuvent surgir entre les différents noms des outils utilisés (prédéfinis ou définis par l'utilisateur), C++ introduit la notion de ***namespace*** (espace de noms), ce qui permet de définir des zones de déclaration et de définitions des différents outils (variables, fonctions, types,…). Ainsi, chaque élément défini dans un programme ou dans une bibliothèque appartient désormais à un namespace. La plupart des outils d'Entrées / Sorties de la bibliothèque de C++ appartiennent à un namespace nommé "***std***".

Syntaxe:

```
using namespace std;
```

Exemple d'utilisation:

```
#include <iostream>
#include <conio.h>
using namespace std;  // Importation de l'espace de nom

void main()
{   double x, y;
    // Le préfixe n'est plus requis :
    cout << "X:";
    cin >> x;
    cout << "Y:";
   // Il est toujours possible d'utiliser le préfixe :
   std::cin >> y;
   cout << "x * y = " << x * y << endl;
_getch();   // Les fonctions de la bibliothèque C sont toujours utilisables
}
```

Il est possible aussi de définir un espace de nom alors les identificateurs de cet espace seront préfixés.

Syntaxe :

```
namespace nom
{
     // Placer ici les déclarations faisant partie de l'espace de nom
}
```

Exemple:

```
#include <iostream>
using namespace std;  // Importation de l'espace de nom

namespace USTOMB
{
   void afficher_adresse()
    {   cout << "USTOMB" << endl;
        cout << "El Mnaouar, BP 1505 , "<<endl<< " Bir El Djir 31000"<<endl ;
    }
```

```
 void afficher_coordonnees_completes()
   {    afficher_adresse();   // Préfixe non requis, car dans le même espace de  nom :
        cout << "Tel :  041 61 71 46\n";
    }
}

int main()
{   USTOMB::afficher_coordonnees_completes();
    return 0;
}
```

II.2.7 Les entrées/sorties en C++

On peut utiliser les routines d'E/S da la bibliothèque standard de C (<stdio.h>). Mais C++ possède aussi ses propres possibilités d'E/S.

Les nouvelles possibilités d'E/S de C++ sont réalisées par l'intermédiaire des opérateurs **<<** (sortie), **>>** (entrée) et des flots (stream) définis dans la bibliothèque <iostream>, suivants [6]:

- ***cin*** : flot d'entrée correspondant à l'entrée standard (instance de la classe ***istream_withassign***)
- ***cout*** : flot de sortie correspondant à la sortie standard (instance de la classe ***ostream_withassign***)
- ***cerr*** : flot de sortie correspondant à la sortie standard d'erreur (instance de la classe ***ostream_withassign***)

Syntaxes :

```
cout << exp_1 << exp_2 << … … << exp_n ;
```

exp_ k : *expression de type de base ou chaîne de caractères*

```
cin >> var_1 >> var_2 >> … … >> var_n ;
```

var_k : *variable de type de base ou* char*

Tous les caractères de formatage comme '\t', '\n' peuvent être utilisés. Par ailleurs, l'expression ***endl*** permet le retour à la ligne et le vidage du tampon.

Exemple:

```
#include <iostream> // Standard C++ I/O
using namespace std;

int main()
{   int val1, val2;
    cout << "Entrer deux entiers: " << endl;
    cin >> val1 >> val2;
    cout << "Valeurs entrées: " << val1 << " et " << val2 << endl;
    cout << "valeur 1+valeur 2 =" << val1 + val2 << endl;
return 0;
}
```

On peut citer quelques avantages des nouvelles possibilités d'E/S :

- Vitesse d'exécution plus rapide.
- Il n'y plus de problème de types
- Autres avantages liés à la POO.

II.2.8 Bibliothèque iomanip

Il existe d'autres possibilités de modifier la façon dont les éléments sont lus ou écrits dans le flot:

- **dec:** Lecture/écriture d'un entier en décimal.
- **oct:** Lecture/écriture d'un entier en octal.
- **hex:** lecture/écriture d'un entier en hexadécimal.
- **endl:** Insère un saut de ligne et vide les tampons.
- **setw(int n):** Affichage de n caractères.
- **setprecision(int n):** Affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur.
- **setfill(char):** Définit le caractère de remplissage flush vide les tampons après écriture.

Exemple:

```
#include <iostream>
#include <iomanip>      // attention a bien inclure cette librairie
using namespace std ;

int main()
{  int i=1234;
  float p=12.3456;
  cout << "|" << setw(8) << setfill('*') << hex << i << "|" <<endl
       << "|" << setw(6) << setprecision(4) << p <<"|"<< endl;
return 0;
}
```

Affichage de l'exécution du code:

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

Process returned 0 (0x0)   execution time : 3.036 s
Press any key to continue.
```

II.3 Variables et constantes

II.3.1 Noms de variables

En C++, il y a quelques règles qui régissent les différents noms autorisés ou interdits [4]:

- Les noms de variables sont constitués de lettres, de chiffres et du tiret-bas « _ » uniquement. Le double souligné "__" au début du nom est réservé aux variables/fonctions du système.
- Le premier caractère doit être une lettre (majuscule ou minuscule).
- On ne peut pas utiliser d'accents.
- On ne peut pas utiliser d'espaces dans le nom.

Exemple:

ageEtudiant, ***nom_etudiant***, ***NOMBRE_Etudiants***: *Noms valides.*

Ageétudiant, ***nom etudiant***: *Noms non valides.*

II.3.2 Types de base

Après l'identification du nom de la variable, L'ordinateur doit connaitre ce qu'il a dans cette variable, il faut donc indiquer quel type (nombre, mot, lettre, ou …) d'élément va contenir la variable que nous aimerions utiliser [4].

Voici donc la liste des types de variables que l'on peut utiliser en C++ (Tableau 1) :

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$-3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant double	8	$-1.7*10^{-308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$-3.4*10^{-4932}$ à $3.4*10^{4932}$
bool	Booléen	Même taille que le type *int*, parfois *1* sur quelques compilateurs	Prend deux valeurs: t*rue* et *false* mais une conversion implicite (valant 0 ou 1) est faite par le compilateur lorsque l'on affecte un

Tableau 1 : Types détaillés de variables en langage C++ [4]

II.3.3 Déclaration des variables

Avant d'utiliser une variable, il faut indiquer à l'ordinateur le type de la variable que nous voulons, son nom et enfin sa valeur c.-à-d. les trois caractéristique de la variable. En effet, en C++, toute variable doit être déclarée avant d'être utilisée.
Une variable se déclare de la façon suivante:

```
type Nom_de_la_variable < (valeur) > ;
```

ou bien utiliser la même syntaxe que dans le langage C.

```
type Nom_de_la_variable < = valeur > ;
```

Soit l'exemple suivant, dans lequel nous déclarons des variables pour stocker les informations d'un étudiant:

```
#include <iostream>
 using namespace std;
 int main()
{  string nomEtudiant(" Mostefa Amine");
   int ageEtudiant(20);
   float  moyGen(12.43);
   double pi(3.14159);
   bool estAdmis(true);
   char lettre('a');
  return 0;
}
```

✍ Pour le type **string**:

- Le C et le C++ n'ont pas de type de base pour représenter les chaînes de caractères (comme en JAVA le type « 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.
- Un peu comme pour le char, il faut mettre la chaine de caractères entre guillemets et non pas entre des apostrophes.

Exemple:

```
#include <iostream>
#include <string>
 using namespace std;
 int main()
{   string nomUtilisateur;
    int nombreamis, a(2), b(4), c(-1);   // réservation de 3 cases mémoires entières avec
                                                                          //initialisation et une sans.
    float  x(2.33), y(21.6);        // déclare deux réels x, y
    string  prenom("Mostefa"), nom("Amine"); // on déclare deux cases pouvant contenir des chaînes
                                                        //  de caractères
    bool ajout; // valeur booléenne sans initialisation
 return 0;
}
```

II.3.4 Constantes

Contrairement aux variables les constantes ne peuvent pas être initialisées après leur déclaration et leur valeur ne peut pas être modifiée après initialisation. Elles doivent être déclarées avec le mot clé **const** et obligatoirement initialisées dès sa définition.

Syntaxe:

```
const <type> <NomConstante> = <valeur>;
```

Exemple:

```
const char c ('A');       //exemple de constante caractère
const int i (2017);       //exemple de constante entière
```

```
const double PI (3.14);    //exemple de constante rèel
const string motDePasse("pass_2017");     //exemple de constante chaine de caractères
```

✍ Alternative pour les constantes de type entier: **Enumérations**

Exemple

```
  enum { Red, Green, Blue, Black, White };
```

et equivaut à

```
  const int Red = 0;
  const int Green = 1;
   ...
  const int White = 4;
```

Une énumération peut-être nommée

```
enum Color { Red, Green, Blue, Black, White };
Color background = Black;
Color foreground;
....
if (background == Black)
foreground = White;
```

II.3.5 Types dérivés

Il existe d'autres dérivés des types fondamentaux et des types définis par le programmeur, cette dérivation se fait à l'aide d'opérateurs de déclaration. On peut distinguer quatre types de dérivation:

- Pointeur
- Référence
- Tableau
- Prototype des fonctions (voir chapitre 3)

a. Pointeurs

Un pointeur contient l'adresse d'un objet en mémoire, l'objet pointé peut être référencé via le pointeur. On peut utiliser les pointeurs pour la manipulation des objets alloués dynamiquement, création et manipulation de structures chaînées, etc.

Exemple:

```
int* ptrInt, q;  // attention ptrInt est un pointeur sur un entier et  q est un entier
double* ptrDouble, *ptr, **data ; // ptrDouble et ptr sont des pointeurs sur des doubles et data
                                  //pointe deux fois sur un double
char *text ;  // text pointe sur un caractère
const char* PCste = "this is a string"; // PConst pointe sur une chaine constante
```

b. Références

En C++, on peut coller plusieurs étiquettes à la même case mémoire (ou variable). On obtient alors un deuxième moyen d'y accéder. On parle parfois d'**alias**, mais le mot correct

en C++ est **référence**. Pour déclarer une référence sur une variable, on utilise une esperluette (**&**).

Exemple:

```
#include <iostream>
using namespace std;
int main()
{  int i(5);
   int &  j(i);        // la variable  'j ' fait référence à 'i '. On peut utiliser à partir d'ici  'j ' ou 'i '
                       // indistinctement puisque ce sont deux étiquettes de la même case en mémoire
   int k=j ;
   cout<<" Valeur de i: "<<i<< " || "<<" Valeur de j: "<<j<<" || "<<" Valeur de k: "<<k<<  endl;
   j=j+2;
   k=i ;
   cout<<" Valeur de i: "<<i<< " || "<<" Valeur de j: "<<j<<" || "<<" Valeur de k: "<<k<<  endl;
return 0;
}
```

Affichage de l'exécution du code:

```
 Valeur de i: 5 ||  Valeur de j: 5 ||  Valeur de k: 5
 Valeur de i: 7 ||  Valeur de j: 7 ||  Valeur de k: 7

Process returned 0 (0x0)   execution time : 3.253 s
Press any key to continue.
```

✍ **Remarque:** Si la variable est définie dans un autre module il faut la déclarer avant de l'utiliser en utilisant le mot clé « **extern** »

Exemple: [1]

```
extern float maxCapacity;  // declaration
float limit = maxCapacity * 0.90;  // usage
```

file1.cpp

```
float maxCapacity;
```

file2.cpp

```
extern float maxCapacity;
float limit = maxCapacity*0.90;
```

c. Tableaux

Un tableau est une collection d'objets du même type, chacun de ces objets est accédé par sa position. La dimension du tableau doit être connue en temps de compilation.

Exemple:

```
const int numPorts = 200;
double portTable[numPorts];
int bufferSize;
char buffer[bufferSize]; // ERROR: the value of bufferSize is unknown
```

Le tableau peut être initialisé lors de la définition :

```
int groupTable[3] = {134, 85, 29};
int userTable[] = {10, 74, 43, 45, 89}; // 5 positions
```

II.3.6 Définition de Nouveaux Types

On peut définir un synonyme pour un type prédéfini à l'aide du mot clé **typedef.**

```
typedef float Angle;
```

Le nouveau type ***Angle*** peut être utilisé pour définir des variables

```
Angle rotation = 234.78;
```

Toutes les opérations valides avec le type original le sont aussi avec le synonyme

```
Angle rotation = "white"; // ERROR: Angle is float not char*
```

II.3.7 Conversion de Type

Ce type d'opération consiste à modifier la façon d'interpréter la séquence de bits contenue dans une variable

a. Conversion implicite

Faite automatiquement par le compilateur (non sans warnings) lorsque plusieurs types de données sont impliqués dans une opération.

- Lors de l'affectation d'une valeur à un objet, le type de la valeur est modifié au type de l'objet

```
int count = 5.890; // conversion à  int 5
```

- Chaque paramètre passé à une fonction est modifié au type de l'argument espéré par la fonction

```
extern int add(int first, int second);
count = add(6.41, 10); // conversion à int 6
```

- Dans les expressions arithmétiques, le type de taille supérieure détermine le type de conversion

```
int count = 3;
count + 5.8978; // conversion à double 3.0 + 5.8978
```

Exemple :

```
int count = 10;
count = count * 2.3;  // Conversion de 23.0 à int 23
```

- Tout pointeur à une valeur non constante de n'importe quel type, peut être affecté à un pointeur de type **void***

```
char* name = 0;
void* aPointer = name; // conversion implicite
```

b. Conversion explicite ou casting

Cette conversion est demandée par le programmeur

Exemple :

```
int result = 34;
result = result + static_cast<int>(10.890);
```

Peut être aussi écrit

```
result = result + int(10.890);
```

Permet d'utiliser des pointeurs génériques

```
char* name;
void* genericPointer;
name = (char*)genericPointer; // Conversion explicite
```

II.4 Expressions [4]

En combinant des noms de variables, des opérateurs, des parenthèses et des appels de fonctions on obtient des expressions.

II.4.1 Opérateurs Arithmétiques et logiques

a. Opérateurs multiplicatifs

Opération	Symbole	Exemple	
Multiplication	*	resultat = a * b;	Si a=5 et b=2 → resultat = 10
Division	/	resultat = a / b;	Si a=5 et b=2 → resultat = 2
Modulo	%	resultat = a % b;	Si a=5 et b=2 → resultat = 1

- Le Modulo représente le reste de la division entière. Cet opérateur n'existe que pour les nombres entiers.
- La division est entière si les deux arguments sont entiers.

b. Opérateurs additifs

Opérateur	Signification	Arité	Associativité
+	Addition unaire	unaire	Non
-	négation unaire	unaire	Non
+	Addition	binaire	Gauche à droite
-	Soustraction	binaire	Gauche à droite

- Il est clair que « l'addition » unaire ne fait rien pour les types simples (nombres entiers et réels), mais grâce à la surcharge des opérateurs, on peut donner un sens à cet opérateur sur des types (=classes) complexes

c. Opérateurs de décalage

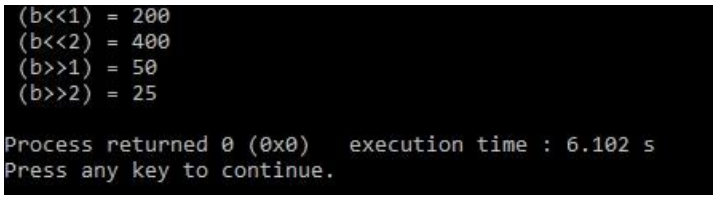
Opérateur	Signification	Arité	Associativité
<<	Décalage à droite	Binaire	Gauche à droite
>>	Décalage à gauche	binaire	Gauche à droite

- Ces opérateurs sont assez peu employés dans leur contexte « C » (décalage de bits), mais ils prennent une importance considérable en C++ dans la manipulation des flux d'entrées/sorties.

Exemple :

```
int b=100 ; // b=1100100 en code binaire
std ::cout<< " (b<<1) = " <<(b<<1) << std ::endl ;
std ::cout<< " (b<<2) = " <<(b<<2) << std ::endl ;
std ::cout<< " (b>>1) = " <<(b>>1) << std ::endl ;
std ::cout<< " (b>>2) = " <<(b>>2) << std ::endl ;
```

Affichage de l'exécution du code:



d. Opérateurs relationnels

Opérateur	Signification	Arité	Associativité
<	Inférieur à	binaire	Gauche à droite
>	Supérieur à	binaire	Gauche à droite
<=	Inférieur ou égal à	binaire	Gauche à droite
>=	Supérieur ou égal à	binaire	Gauche à droite

- en C++, ces opérateurs retournent un type **bool** (valeurs **true** ou **false**) contrairement au C où ils retournent un type **int** (valeurs **0** ou **1**)

e. Opérateurs d'égalité

Opérateur	Signification	Arité	Associativité
==	Egalité	binaire	Gauche à droite
!=	Inégalité	binaire	Gauche à droite

- Erreur classique: ne pas confondre l'opérateur d'égalité (==) avec l'opérateur d'affectation (=).

f. Opérateurs sur les bits

Opérateur	Signification	Arité	Associativité
&	Et binaire	binaire	Gauche à droite
^	Ou exclusif binaire	binaire	Gauche à droite
|	Ou binaire binaire	binaire	Gauche à droite

- Ces opérateurs travaillent sur les bits, comme les décalages. Attention de ne pas les confondre avec les opérateurs logiques.

g. Opérateurs logiques

Opérateur	Signification	Arité	Associativité
&&	Et logique	binaire	Gauche à droite
||	Ou logique	binaire	Gauche à droite
e1 ?e2 :e3	Condition if - else	ternaire	De droite à gauche

- Ces opérateurs travaillent sur le type **bool** et retournent un **bool.**
- Le dernier opérateur signifie: « **si** e1 est vraie **alors** faire e2, **sinon** faire e3 »

h. Opérateurs d'affectation

Opérateur	Signification	Arité	Associativité
=	Affectation	R = a	Droite à gauche
*=	Multiplication et affectation	a = a*b	Droite à gauche
/=	Division et affectation	a = a/b	Droite à gauche
%=	Modulo et affectation	a = a%b	Droite à gauche

+=	Addition et affectation	a = a+b	Droite à gauche
-=	Soustraction et affectation	a = a-b	Droite à gauche
<<=	Décalage à gauche et affectation	a = a<<b	Droite à gauche
>>=	Décalage à droite et affectation	a = a>>b	Droite à gauche
&=	Et binaire et affectation	a = a&b	Droite à gauche
|=	Ou binaire	a = a|b	Droite à gauche
^=	Ou exclusif binaire et affectation	a = a^b	Droite à gauche
++	Incrémentation	a = a+1 ≈ a++	Droite à gauche
--	Décrémentation	a = a-1 ≈ a--	Droite à gauche

Exemple:

```
int a, b=3, c, d=3 ;
a = ++b ;  // équivalent à b++ ; puis a=b ; => a=b=4
c = d++ ;  // équivalent à c=d ; puis d++ ; => c=3 et d=4
```

II.5 Structures de contrôle

II.5.1 Structures de contrôle conditionnelles

On appelle structures de contrôle conditionnelles les instructions qui permettent de tester si une condition est vraie ou non. Il s'agit des instructions suivantes:

- La structure conditionnelle **if**
- Le branchement conditionnel **switch**

II.5.1.1 Structure conditionnelle « if »

a. Première condition if

La structure conditionnelle **if** permet de réaliser un test et d'exécuter une instruction ou non selon le résultat de ce test [4a].

Syntaxe :

```
if (condition)
{  instruction ;  }
```

où condition est une expression dont la valeur est booléenne ou entière. Toute valeur non nulle est considérée comme vraie. Si le test est vrai, instruction est exécutée.

- Il est possible de définir plusieurs conditions à remplir avec les opérateurs ET et OU (**&&** et **||**). Par exemple, l'instruction ci-dessous exécutera les instructions si l'une ou l'autre des deux conditions est vraie :

Exemple:

```
if ((condition1) || (condition2))
 {  instruction ;  }
```

b. if … else : *ce qu'il faut faire si la condition n'est pas vérifiée*

L'instruction **if** dans sa forme basique ne permet de tester qu'une condition, or la plupart du temps on aimerait pouvoir choisir les instructions à exécuter en cas de non réalisation de la condition. L'expression **if ... else** permet d'exécuter une autre série d'instructions en cas de non-réalisation de la condition.

Syntaxe :

```
if (condition)
{ instruction ;  }
else
 { autre instruction ;  }
```

c. else if : *effectuer un autre test*

Il est possible de faire plusieurs tests à la suite. Pour faire tous ces tests un à un dans l'ordre, on va avoir recours à la condition ***else if*** qui signifie « sinon si ». Les tests vont être lus dans l'ordre jusqu'à ce que l'un d'entre eux soit vérifié.

Exemple:

```
#include <iostream>
using namespace std;
int main()
{  float a;
  cout<<"un réel : "; cin>>a ;
 if(a>0)
  cout<< a << " est positif " <<  endl;
 else
    if(a==0)
      cout<< a << " est nul " <<  endl;
   else
      cout<< a << " est négatif " <<  endl;
return 0;
}
```

II.5.1.2 Branchement conditionnel switch

Dans le cas où plusieurs instructions différentes doivent être exécutées selon la valeur d'une variable de type intégral, l'écriture des ***if*** successifs peut être relativement lourde. Le C++ fournit donc la structure de contrôle ***switch***, qui permet de réaliser un branchement conditionnel.

Syntaxe :

```
switch (choix)
{
   case valeur1 :
            instruction1;
            break;
   case valeur2 :
            instruction2;
            break;
  case valeur3 :
          instruction3;
          break;
  default:
          instructionParDéfaut;
 }
```

La variable ***choix*** est évaluée en premier. Son type doit être entier. Selon le résultat de l'évaluation, l'exécution du programme se poursuit au cas de même valeur. Si aucun des cas ne correspond et si default est présent, l'exécution se poursuit après ***default***. Si en revanche default n'est pas présent, on sort du ***switch***. Pour forcer la sortie du switch, on doit utiliser le mot-clé ***break***.

Exemple: [4a]

```
#include <iostream>
using namespace std;
int main()
{  int a;
   cout << "Tapez la valeur de a : ";
   cin >> a;                                  // Ce programme demande à l'utilisateur de taper une
   switch(a)                                  // valeur entière et la stocke dans la variable a. On teste
   {  case 1 :                                // ensuite la valeur de a : en fonction de cette valeur on
         cout << "a vaut 1" << endl;          // affiche respectivement les messages "a vaut 1","a vaut 2
         break;                              // ou 4","a vaut 3, 7 ou 8", ou "valeur autre".
      case 2 :
      case 4 :
         cout << "a vaut 2 ou 4" << endl;
         break;
      case 3 :
      case 7 :
      case 8 :
         cout << "a vaut 3, 7 ou 8" << endl;
         break;
      default :
         cout << "valeur autre" << endl;
   }
   return 0;
}
```

Affichage de l'exécution du code:

```
Tapez la valeur de a : 1
a vaut 1

Process returned 0 (0x0)   execution time : 3.355 s
Press any key to continue.
```

```
Tapez la valeur de a : 3
  a vaut 3, 7 ou 8

Process returned 0 (0x0)   execution time : 32.037 s
Press any key to continue.
```

II.5.2 Structures de contrôle itératives [4a]

Les structures de contrôle itératives sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus réalisée. On appelle ces structures des boucles.

Il existe 3 types de boucles **à connaître** :

- la boucle **for**,
- la boucle **while** et
- la boucle **do … while**.

II.5.2.1 Boucle for

La structure de contrôle ***for*** est sans doute l'une des plus importantes. Elle permet de condenser:

- ***Un compteur***: une instruction (ou un bloc d'instructions) exécutée avant le premier parcours de la boucle du for. Il consiste à préciser le nom de la variable qui sert de compteur et éventuellement sa valeur de départ.
- ***Une condition***: une expression dont la valeur déterminera la fin de la boucle.
- ***Une itération***: une instruction qui incrémente ou décrémente le compteur.

Syntaxe:

```
for (compteur; condition; itération))
 {
   Iinstructions ;
}
```

Exemple :

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

int main()
{  int compteur(0);
   for (compteur = 1 ; compteur < 10 ; compteur++)
     cout << compteur <<" || " ;
  cout << endl ;
return 0 ;
}
```

Affichage de l'exécution du code:

```
1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 ||

Process returned 0 (0x0)   execution time : 1.678 s
Press any key to continue.
```

II.5.2.2 Boucle while

Le ***while*** permet d'exécuter des instructions en boucle tant qu'une condition est vraie.

Syntaxe:

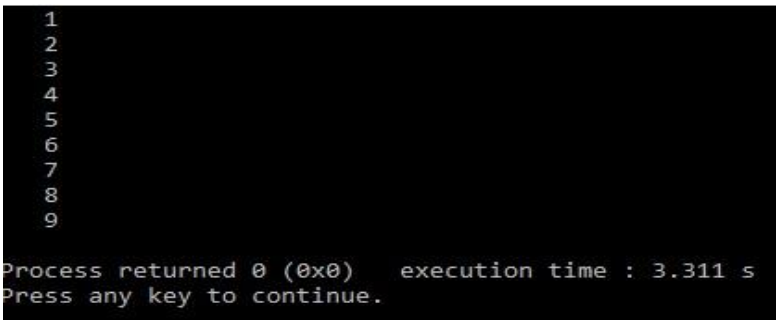
```
while (condition)
{ instructions ; }
```

Exemple :

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

int main()
{  int i = 1;
   while (i < 10)       //affiche les valeurs de 1 jusqu'à 9
     {  cout <<"  "<< i <<endl ;
        i++ ;
     }
 return 0 ;
}
```

Affichage de l'exécution du code:



II.5.2.3 Boucle do … while

La structure de contrôle ***do … while*** permet, tout comme le while, de réaliser des boucles en attente d'une condition. Cependant, contrairement à celui-ci, le do … while effectue le test sur la condition après l'exécution des instructions. Cela signifie que les instructions sont toujours exécutées au moins une fois, que le test soit vérifié ou non.

Syntaxe:

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

Exemple:

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

 int main()
{  int i = 1;
   do
   {  cout << i << endl;
      i++;
   } while(i<10);       //affiche les valeurs de 1 jusqu'à 9
 return 0;
}
```

II.5.3 Ruptures de Séquence

II.5.3.1 break

L'instruction ***break*** sert à "casser" ou interrompre une boucle (for, while et do … while), ou un switch. L'exécution reprend immédiatement après le bloc terminé.

Exemple:

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

int main()
{ int i;
  for (i = 0 ; i < 10 ; i++)
   { cout <<" i= " <<i<< endl;
     if (i==3)
       break;
   }
cout<<" Valeur de i lors de la sortie de la boucle : " << i <<endl ; return 0;
}
```

Affichage de l'exécution du code:

```
i= 0
i= 1
i= 2
i= 3
Valeur de i lors de la sortie de la boucle : 3

rocess returned 0 (0x0)   execution time : 3.386 s
ress any key to continue.
```

II.5.3.2 continue

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

Exemple:

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

int main()
{ int i;
  for (i = 0 ; i < 5 ; i++)
   { if (i==3)
       continue;
     cout <<" i= " <<i<< endl;
   }
cout<<" Valeur de iaprès la sortie de la boucle : " << i <<endl ; return 0;
}
```

Affichage de l'exécution du code:

```
i= 0
i= 1
i= 2
i= 4
Valeur de iaprPs la sortie de la boucle : 5

Process returned 0 (0x0)   execution time : 3.007 s
Press any key to continue.
```