CM – Programmation C/C++

Chapitre 1 : Notions de base

I- Introduction :

Qu'est-ce que la programmation ?

La programmation ne sert pas à naviguer sur Internet, créer des tableurs ou jouer à des jeux vidéo. Il s'agit de donner des instructions à un ordinateur pour qu'il réalise les tâches que l'on souhaite. Pour cela, on utilise un langage de programmation, possédant des commandes (du vocabulaire) et une syntaxe (des phrases, une grammaire, une ponctuation). Le C++ est donc un langage comme le Français ou l'Anglais.

Comment ça marche ?

Un ordinateur ne comprend que les 0 ou les 1, donc on pourrait programmer avec ça (ce qui est complexe) ou utiliser un traducteur pour se faire. En C++, on écrit un fichier texte appelé « code source ». Il sera transformé en langage machine (0 et 1, aussi appelé binaire) par un compilateur. Lorsqu'un code est composé de plusieurs fichiers, le compilateur permet aussi de les relier (ou linker) en un exécutable.

Un logiciel de codage, Code::Blocks

Pour coder dans cette UE, on utilisera Code::Blocks, un environnement de développement intégré pour C++. Il possède un éditeur de texte, un compilateur et un linker entre autres. Il est gratuit et fonctionne sous Linux, Windows et iOS.

II- Les principaux ingrédients d'un programme :

Pour créer un programme en Informatique, on manipule des informations que l'on stocke dans des variables. Un programme est organisé, structuré par des blocs et des fonctions (dont une principale). Il doit pouvoir recevoir les informations de départ pour donner des résultats : ce sont des entrées / sorties (ou input / output). Des outils permettent de manipuler les différentes variables.

1) Les variables :

Une variable est de la forme :

```
type Nom;
type Nom = valeur;
type Nom(valeur);
```

On peut initialiser une variable sans valeur prédéfinie (cas 1) ou avec une valeur (cas 2 ou cas 3, les deux sont possibles). Le point-virgule sert de point pour terminer une ligne, il est donc très important. Elle possède un type correspondant à la valeur qu'on souhaite lui attribuer (entier, réel, caractère, autre…) et un nom qui l'identifie.

Son nom doit être écrit à l'anglaise, donc sans accents. Il peut être composé de lettres, majuscules et minuscules car elles sont différenciées, de chiffres et du caractère « _ », en plus de commencer par une lettre ou « _ ». Il ne peut pas excéder 247 caractères et ne peut être un élément réservé du langage C++.

2) Les types :

Le type d'une variable permet de définir la valeur qu'elle contiendra. Il en existe trois principaux : les entiers (valeurs chiffrées sans virgule), les réels (valeurs chiffrées avec virgule) et les booléens (valeur binaire, donc ne prenant que deux valeurs).

Si le type de variable est un entier, on peut utiliser les mots suivants :

Type de donnée	Nombre d'octets codés	Plage de valeurs
int, long	4	- 2147483648 à + 2147483647
short	2	- 32768 à + 32767
char	1	- 128 à + 127
unsigned int, long	4	0 à 4294967295
unsigned short	2	0 à 65535
unsigned char	1	0 à 255

Les types **`int`**, **`long`** et **`short`** sont des types qui sont liés à des nombres entiers. Le type **`char`**, venant de « character » (caractère), permet de coder tous les caractères que possède un clavier. Les différences en termes d'octets codés par chaque type permettent de prendre une place dans la mémoire différente, ce qui est intéressant selon le programme que l'on souhaite créer. Si un type est **`unsigned`**, la plage de valeurs qu'il possède est strictement positive et va de 0 au double de sa valeur positive maximale de base.

Si le type de variable est un réel, on peut utiliser les mots suivants :

Type de donnée	Nombre d'octets codés	Précision de la variable
float	4	7 décimales
double	8	15 décimales
long double	10	19 décimales

Les types **`float`**, **`double`** et **`long double`** se différencient par le nombre de décimales qu'ils possèdent, impliquant une place dans la mémoire plus ou moins importante.

Si le type de variable est un booléen, on utilise le mot **`bool`**. La variable ne pourra prendre que deux valeurs : **`true`** (signifiant « vrai », valant 1) ou **`false`** (signifiant « faux », valant 0).

3) Les types particuliers :

En plus des types principaux, il existe deux types particuliers pour définir des variables : le type **`void`** et le type **`auto`**.

Le type **`void`** (vide) qui s'utilise quand la syntaxe impose de préciser un type mais qu'il n'y en a pas. On ne peut donc pas déclarer de variable de ce type, mais il servira dans d'autres circonstances.

Le type **`auto`**, permettant au compilateur de définir lui-même le type de la variable en fonction de l'affectation :

```
auto unEntier=2;
auto unDouble=2.0;
```

Dans l'exemple ci-dessus, son utilisation est problématique car, dans le cas de la variable `unDouble`, on ne sait pas si on a un **`double`**, un **`float`** ou un **`long double`**. Un cas où cela est plus efficace est dans le cas où l'on initialise une variable dont la valeur est celle d'une autre variable déjà initialisée et de type connu :

```
int a=3;
auto b=a;
```

Ce type est utile quand le type à affecter n'est pas ambigu. Il vaut mieux l'utiliser avec des types complexes définis par l'utilisateur (structures, classes…).

4) Les entrées / sorties :

Les entrées / sorties (input / output) sont représentées par des flux (stream) qui sont entrants ou sortants. Leur destination et leur provenance sont variées, comme des fichiers, du matériel électronique ou même un autre ordinateur. Dans les flux les plus courants, on a le flux sortant vers l'affichage dans une fenêtre :

```
cout << "bonjour";
```

(avec **cout**, prononcé « c out », qui représente l'écran sur lequel sera affiché le message et << l'opérateur de flux sortant. Les guillemets autour du mot bonjour signifie que c'est un message à afficher et qu'il ne faut pas que le compilateur le traduise).

On a aussi le flux entrant en provenance du clavier de l'ordinateur pour entrer une valeur dans une variable :

```
int a;
cin << a;
```

(avec **cin**, prononcé « c in », qui représente le clavier qui permet d'entrer la valeur et >> l'opérateur de flux entrant).

5) Exemple de programme :

Dans cet exemple, on créé un programme nommé « exemple1.cpp ». Le programme est le suivant :

```
#include <iostream>

using namespace std;


int main()
{
    cout << "Mon 1er programme cpp" << endl;

    return 0;
}
```

Le programme, après compilation, ouvrira un invité de commandes dans lequel il affichera la phrase « Mon 1er programme cpp ».

Dans le code, la première ligne correspond à la direction de compilation, contenant les instructions qui peuvent être inclues depuis un autre fichier en écrivant `#include <fichier>` (le fichier en question étant une librairie). On écrit ensuite **`using namespace std;`** pour créer un nom d'espace pour le programme. La fonction principale est la fonction **`int`** `main()` contenant les instructions du programme, comme celle qui affiche le texte. Elle est délimitée par des accolades qui créent un bloc. Le mot **`endl`** signifie « end line » et sert de retour à la ligne. Elle se finit par **`return`** `0;` pour arrêter la lecture des instructions et terminer le programme.

Chaque programme doit contenir une seule fonction `main()`. Les éléments du code sont colorés pour une meilleure lisibilité et reconnaissance.

6) Les commentaires :

Les commentaires sont des parties du code que le compilateur ignore. Ils permettent d'aider à mieux comprendre le programme car on peut expliquer ce qui est réalisé à une ligne précise. Ils sont écrits en gris et on utilise `/*  */` pour encadrer le message servant de commentaire (il peut s'écrire sur plusieurs lignes) ou `//` à la fin d'une ligne mais il faut le répéter quand on change de ligne. Un bon programme est donc bien commenté.

7) Deuxième exemple de programme :

Pour ce second exemple, on souhaite écrire un programme permettant d'afficher à l'écran :

Mon 2ème programme C++

C++ est facile

Le programme est le suivant :

```
#include <iostream>

using namespace std;


int main() // Fonction principale
{    /* Ecriture à l'écran */
     cout << "Mon 2eme programme C++" << endl;
     cout << endl;
     cout << "C++ est facile";

     return 0;
}
```

Dans ce programme, on aurait pu directement écrire `<< endl;` à la fin de la première instruction pour gagner de la place.

III- Les opérateurs :

1) Notions de base :

Les opérateurs permettent de réaliser une action qui prend plusieurs opérandes pour retourner un résultat. Il en existe plusieurs comme les opérateurs binaires (à deux opérandes) ou unaires (à un opérande).

Les opérateurs binaires se mettent entre deux opérandes et ils retournent une valeur. Ils peuvent être symétrique (ils ne dépendent pas du sens des opérandes) ou non. Des exemples connus sont l'opérateur `+` (somme arithmétique classique : `3.1+5.4` retournera `8.5`) ou `=` (affectation d'une valeur à une variable, donc différent de son sens mathématique). Dans le cas de l'opérateur « = », on peut prendre un exemple pour illustrer son fonctionnement :

```
int heure;
heure=18
```

Ici, on crée une variable nommée `heure` et on lui affecte la valeur 18. Cependant, inverser les positions des deux opérandes fait que rien ne marche : `18 = heure;` n'affecte rien, l'opérateur `=` est donc non symétrique.

Les opérateurs unaires se mettent avant ou après l'opérande choisi et renvoient aussi une valeur. Cette catégorie est composée notamment des opérateurs incréments `++` (on ajoute 1 à l'opérande, ce qui équivaut à écrire : `i=i+1;`).

2) Les opérateurs d'affectation :

L'opérateur `=` est l'opérateur d'affectation, permettant de donner une valeur à une variable. On peut le combiner avec d'autres opérateurs, par exemple en affectant à une variable la valeur d'une somme :

```
int heure;
heure=12+1;
```

Ici, on fait la somme `12 + 1` qui retourne 13 et on l'associe à la variable `heure`. L'évaluation de la combinaison se fait donc de la droite vers la gauche.

On peut aussi affecter une même valeur à plusieurs variables en une seule expression :

```
int i;
int j;
i=j=1024;
```

Ici, on initialise deux variables `i` et `j`, puis on affecte la valeur `1024` à `j` et, après cela, on affecte `j` à `i` donc la valeur de `j` est affectée à `i`.

3) Les opérateurs arithmétiques :

Les opérateurs `+` (somme), `-` (différence), `/` (division) et `*` (multiplication) sont des opérateurs arithmétiques, fonctionnant de la même manière que pour les Mathématiques. Le résultat d'une de ces opérations dépend du type des opérandes : si les deux opérandes sont entiers, le résultat est un entier ; si les deux opérandes sont réels, le résultat est un réel ; si l'un des opérandes est entier et l'autre réel, le résultat est un réel.

La division possède la particularité d'être entière si les deux opérandes sont entiers. Si on fait la division `3 / 2;`, elle retournera 1, alors que si on fait la division `3.0 / 2;` (l'ajout du point entre le 3 et le 0 indique que l'opérande est réel), elle retournera 1.5, définissant une division réelle.

Dans les opérateurs arithmétiques particuliers, on a :

- L'opérateur `%` (modulo) : `c = a % b;`, qui permet de garder le reste de la division euclidienne de l'opérande de gauche par l'opérande de droite. Cet opérateur est à utiliser uniquement que pour des entiers.
- Les opérateurs `=+` et `=-` (plus et moins unaire) : `b =+ a;` et `b =- a;`, qui permet d'affecter à l'opérande de gauche la valeur signée positive ou négative de l'opérande de droite.
- L'opérateur `++` (incrément) qui dépend de la position de l'opérande associé : `b = a++;` est un post-incrément qui permet d'affecter à l'opérande de gauche la valeur de l'opérande de droite, puis d'incrémenter (ajouter 1) la valeur de l'opérande de droite ; `b = ++a;` est un pré-incrément qui permet d'incrémenter l'opérande de droite, puis d'affecter à l'opérande de gauche la valeur de l'opérande de droite.
- L'opérateur `--` (décrément) qui fonctionne comme l'incrément, excepté qu'il soustrait 1 à la place : `b = a--;` et `b = --a;`.

4) Les opérateurs combinés :

Les opérateurs combinés sont des raccourcis composés d'un opérateur arithmétique avec un opérateur d'affectation. Cela évite d'écrire l'intégralité d'une opération comportant les deux opérateurs si l'un des deux opérandes de l'opérateur arithmétique est le même que celui à qui on associe l'opération : `a += b;` signifie `a = a + b;`. Cela fonction pour tous les opérateurs uniques cités précédemment (`+`, `-`, `/`, `*` et `%`).

5) *Troisième exemple de programme :*

Dans ce troisième exemple, on souhaite écrire un programme qui demande à l'utilisateur de rentrer deux entiers puis d'en afficher la somme. Le programme est le suivant :

```
#include <iostream>

using namespace std;


int main()
{
    cout << "Entrer un entier :";
    int a;
    cin >> a;

    cout << "Entrer un 2eme entier :";
    int b;
    cin >> b;

    cout << "Somme=" << a+b << endl;

    return 0;
}
```

6) *Les opérateurs relationnels et logiques :*

Les opérateurs relationnels et logiques permettent de faire des comparaisons entre deux opérandes et de retourner un booléen, donc soit **`true`** (vrai), soit **`false`** (faux), si la condition donnée par l'opérateur est bonne ou non.

Les opérateurs relationnels sont :

- L'opérateur `==` (égalité) qui compare deux opérandes et renvoie **`true`** s'ils sont identiques ou **`false`** s'ils sont différents. Cet opérateur n'est pas le même que `=` car ce dernier affecte la valeur de l'opérande de droite à l'opérande de gauche alors que le premier ne fait que comparer les deux sans changer leurs valeurs.
- L'opérateur `!=` (inégalité) qui compare deux opérandes et renvoie **`true`** s'ils sont différents ou **`false`** s'ils sont identiques.
- L'opérateur `<` (inférieur) qui compare deux opérandes et renvoie **`true`** si la valeur de celui de gauche est strictement inférieure à la valeur de celui de droite, **`false`** sinon.
- L'opérateur `<=` (inférieur ou égal) qui compare deux opérandes et renvoie **`true`** si la valeur de celui de gauche est inférieure ou égale à la valeur de celui de droite, **`false`** sinon.
- L'opérateur `>` (supérieur) qui compare deux opérandes et renvoie **`true`** si la valeur de celui de gauche est strictement supérieure à la valeur de celui de droite, **`false`** sinon.

- L'opérateur `>=` (supérieur ou égal) qui compare deux opérandes et renvoie **`true`** si la valeur de celui de gauche est supérieure ou égale à la valeur de celui de droite, **`false`** sinon.

Les opérateurs logiques ont, comme particularité, d'avoir deux opérandes de type booléen. Ils sont souvent composés avec les opérateurs relationnels car ils retournent un booléen. Ces opérateurs sont :

- L'opérateur `&&` (ET logique) qui renvoie **`true`** si les deux conditions sont vraies, donc renvoies **`true`**. Si l'une des deux conditions (opérandes) ou les deux ne sont pas vraies, l'opérateur renvoie **`false`**.
- L'opérateur `||` (OU logique) qui renvoie **`true`** si l'une des deux conditions est vraie, donc renvoie **`true`**. Si les deux conditions sont vraies ou si les deux conditions sont fausses, l'opérateur renvoie **`false`**.
- L'opérateur `!` (négation) qui renvoie **`true`** si la condition qu'il contient n'est pas vraie, donc renvoie **`false`**. Sinon, l'opérateur renvoie **`false`**.

IV- L'instruction de sélection :

1) Instructions « si » et « sinon » :

L'instruction de sélection **`if`** (« si » en Anglais) permet, en lui donnant une condition spécifique, de réaliser une instruction précise si la condition est vraie. Dans le cas où la condition n'est pas vraie, l'instruction est passée et le programme continue. Dans le cas simple d'une instruction, on l'écrit :

```
if(condition)
    instruction;
```

Dans le cas de plusieurs instructions à donner, on les écrit entre des accolades :

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

On peut aussi donner des instructions dans le cas où la condition n'est pas remplie avec l'instruction de sélection **`else`** (« sinon » en Anglais). On écrit cette instruction après celle du **`if`** :

```
if(condition)
    instruction;
else
    instruction;
```

Dans le cas de plusieurs instructions à donner, on réitère comme pour juste **`if`** :

```
if(condition)
{
    instructions;
}
else
{
    instructions;
}
```

Exemple – Programme renvoyant si un nombre est positif ou négatif :

```
#include <iostream>

using namespace std;


int main()
{
    double a;
    cout << "Entrer un nombre réel :" << endl;
    cin >> a;

    if(a<0)
        cout << a << "est négatif" << endl;
    else
        cout << a << "est positif" << endl;

    return 0;
}
```

2) Switch :

Lorsqu'il y a un grand nombre d'instructions de sélections, on peut utiliser un **`switch`** à la place de mettre plusieurs **`if`**. On écrit alors :

```
switch(variable)
{
    case val1 :
    {
        instructions;
        break;
    }

    case val2 :
    {
        instructions;
        break;
    }

    default :
    {
        instructions;
        break;
    }
}
```

Un **switch** prend une `variable` quelconque en entrée et se divise en différents cas, notés **case**, définis par une valeur `val1`, `val2`, etc…, contenant des instructions et qui sont réalisées si la valeur de `variable` est égale à la valeur d'un des cas. Si aucune valeur ne convient à un cas, on applique les instructions dans le cas **default**.

Exemple – Programme affichant en lettre un entier inférieur à 5 entré par l'utilisateur :

```
#include <iostream>

using namespace std;


int main()
{
    int unEntier;
    cout << "Entrer un entier < 5 :" << endl;
    cin >> unEntier;
```

```
int main()
{
    int unEntier;
    cout << "Entrer un entier < 5 :" << endl;
    cin >> unEntier;

    switch(unEntier)
    {
        case 1 :
        {
            cout << "Un" << endl;
            break;
        }
        case 2 :
        {
            cout << "Deux" << endl;
            break;
        }
        case 3 :
        {
            cout << "Trois" << endl;
            break;
        }
        case 4 :
        {
            cout << "Quatre" << endl;
            break;
        }
        default :
            cout << "Mauvais choix" << endl;
    }

    return 0;
}
```

V- Les boucles :

Les boucles permettent de réaliser une même séquence plusieurs fois à l'ordinateur. Il en existe trois types de boucles : **`while`**, **`do while`** et **`for`**. On sort de ces boucles soit quand elles sont finies, soit avec des opérateurs de rupture : **`break`** et **`continue`**.

1) La boucle while :

La boucle **`while`** (« tant que » en anglais) permet, tant que la condition est vraie, d'exécuter les instructions à l'intérieur. Cette boucle peut se répéter un nombre aléatoire de fois jusqu'à ce que la condition soit fausse. La boucle évalue directement la condition en début, donc elle ne se fait pas si elle est fausse dès le départ. On écrit cette boucle :

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

Exemple – Programme comptant de 0 à 100 :

```
#include <iostream>

using namespace std;


int main()
{
    int i=1;

    while(i=100)
    {
        cout << i << endl;
        i++;
    }

    return 0;
}
```

2) La boucle do while :

La boucle **`do while`** (« faire tant que » en anglais) est similaire à la boucle **`while`** excepté qu'elle permet d'exécuter les instructions du programme une première fois avant de vérifier les conditions à la toute première itération de la boucle.

La partie de la boucle qui contient les instructions sera dans des accolades après le **do** et on écrit ensuite le **while** avec sa condition :

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

Exemple – Programme demandant à l'utilisateur un entier entre 0 et 100 et vérifiant s'il est bien entre 0 et 100, sinon il recommence :

```
#include <iostream>

using namespace std;


int main()
{
    int i;

    do {
        cout << "Entrer un entier entre 0 et 100" << endl;
        cin >> i;
    } while(i>100 || i<0);

    return 0;
}
```

3) La boucle for :

La boucle **for** (« pour » en anglais) permet d'initialiser une variable, appelée `expression` qui lui est interne et qui fait office de compteur pour le nombre de répétitions de la boucle. Elle commence par initialiser `expression`, puis si la condition est **true**, les instructions sont exécutées avant que `expression` ne le soit, puis on revérifie la condition. On l'écrit :

```
for(init_expression;condition;expression)
{
    instructions;
}
```

Exemple – Programme comptant de 0 à 100 :

```
#include <iostream>

using namespace std;


int main()
{
    for(int i=1;i<=100;i++)
        cout << i << endl;

    return 0;
}
```

4) Les instructions de rupture :

Les instructions de rupture permettent de mettre fin à une boucle prématurément à l'endroit où elles sont écrites. Les instructions de rupture sont :

- **`break`** termine l'exécution de la boucle (ou aussi du switch) la plus interne. L'exécution du programme reprend directement après le bloc, passant le reste de la boucle.
- **`continue`** termine l'exécution l'itération de la boucle la plus interne, puis on repart directement au début de la boucle. Le programme continue donc à exécuter la boucle, mais passe l'itération souhaitée.

VI- Les tableaux :

1) Définition :

Un tableau est une collection de variables du même type (à ne pas voir comme un tableau de nombres classique mais plus comme une liste). Un tableau s'écrit de la façon suivante :

```
type Nom[n];
```

Il est composé du type de chacun des éléments du tableau, d'un nom qui le définit et d'un nombre `n` qui est le nombre d'éléments du tableau (il doit être strictement positif). Un tableau sans nombre d'éléments ne peut être créé.

2) Indexation :

Les éléments d'un tableau sont indexés à partir de 0, c'est-à-dire que si le tableau possède 10 éléments, la numérotation de ses cases ira de 0 à 9. Le compilateur ne garantit pas le contrôle de la validité d'une position dans le tableau (si on ajoute une valeur indexée en 12 dans le tableau en 10, on fait une erreur d'exécution, mais pas de compilation, c'est-à-dire que le code pourra être compilé normalement mais le programme va crasher à son lancement).

3) *Initialisation :*

Lors de l'initialisation, on peut écrire les valeurs de chaque case du tableau dès le départ de ces deux manières :

```
int Tab[3]={56,38,199};
int Tab2[]={156,76,9,0};
```

Dans le premier cas, on créé un tableau à 3 éléments et on lui donne trois valeurs différents en mettant un opérateur = et des accolades avec les valeurs séparées par des virgules. Dans le second cas, le tableau n'a pas de nombre d'éléments mais le fait que l'on ait mis les valeurs de chaque case à la place permet de faire comprendre au programme que la tableau est de la même taille que le nombre de valeurs initialisées.

Les tableaux sont souvent utilisés avec une boucle **for** car elle permet, dans ses conditions, de passer sur toutes les valeurs du tableau sans aller dans une case inexistante.

4) *Tableaux à plusieurs dimensions :*

Un tableau à plusieurs dimensions est comme un tableau classique avec des cases qui forment différentes colonnes et lignes. Pour ajouter des dimensions à un tableau, on rajoute des paires de crochets pour autant de dimensions que l'on souhaite ajouter. Pour initialiser un tableau à plusieurs dimensions, on peut le faire de trois façons différentes :

- On initialise toutes les valeurs en une ligne comme pour le tableau à une dimension : `int A[3][2] = { {1,2}, {8,-1}, {6,19} };`.
- On initialise le nombre d'éléments de la seconde dimension mais pas de la première tout en écrivant les valeurs de la même manière que la première option car le programme comprendra qu'il y a une dimension à 3 valeurs : `int A[][2] = { {1,2}, {8,-1}, {6,19} };`.
- On écrit chaque valeur manuellement pour chaque case, ce qui est long : `A[0][0] = 1; A[1][0] = 8; A[0][1] = 2;`, etc…

VII- Les chaînes de caractères en C :

1) Description :

Dans les types présentés précédemment, il y a le type **char** qui permet d'écrire des chaînes de caractère avec des caractères alphanumériques.

Sa valeur s'écrit entre apostrophes :

```
char c='a';
```

Il existe des caractères spéciaux comme '\n' pour un retour à la ligne, '\t' pour une tabulation et '\a\' pour un bip sonore.

Une chaîne de caractères (mot, phrase, etc.) est un tableau de caractères. On peut initialiser les valeurs de ce tableau sans avoir à mettre toutes les valeurs comme un tableau classique. Par exemple, pour écrire le mot toto, on peut écrire «toto» au lieu d'écrire {'t','o','t','o'}.

Ce fonctionnement est valable que pour le C, mais en C++ on peut fournir un nouveau type pour les chaînes de caractères : le type **string** (c'est aussi une classe du C++). Un objet de type **string** est un tableau de **char** amélioré qui inclut des fonctions permettant de faciliter le traitement des chaînes (comparaison, recherche, concaténation, suppression, etc.). Ce type appartient à une bibliothèque qu'il faut inclure au programme, la bibliothèque <string>, et il s'utilise comme n'importe quelle fonction.

2) Opérateurs des strings :

On peut utiliser les opérateurs relationnels sur les chaînes de caractères comme d'autres objets d'autres types, mais les effets de ces opérateurs sont légèrement différents :

- L'opérateur == (égalité) permet de savoir si deux chaînes sont identiques.
- L'opérateur != (inégalité) permet de savoir si deux chaînes sont différentes.
- L'opérateur < (inférieur) permet de savoir si la première chaîne est devant la seconde par ordre alphabétique (cet ordre suit l'ordre du code ASCII, donc les chiffres sont plus petits que les lettres, les majuscules plus petites que les minuscules, etc…). On peut aussi utiliser >, <=, >=.
- L'opérateur + permet de concaténer deux chaînes, c'est-à-dire que l'on prend la première chaine de caractères et on ajoute à sa fin la seconde.

On ne peut pas utiliser -, * ou / car ce sont des opérateurs mathématiques.

VIII- Structure du programme en blocs :

En C++, les variables existent localement dans les blocs et sous-blocs où elles sont créées. Un bloc est défini par des accolades `{}` à son début et à sa fin. Un sous-bloc est défini de la même manière mais il existe dans un bloc.

À la sortie d'un bloc, toutes les variables qui y ont été créé sont effacées. Le nom d'une variable doit être unique au sein d'un même bloc, mais il peut exister deux variables à deux noms identiques dans deux blocs différents (ou un bloc et un sous-bloc). Dans ce cas, la variable prise en compte est celle du bloc le plus inférieur.

IX- La fonction rand() :

La fonction `rand()` est une notion qui sort un peu des notions de base. Elle fait partie de la bibliothèque `<cstdlib>` et contient un générateur de nombres aléatoires. Chaque appel de cette fonction renvoie un nouveau nombre aléatoire entier compris entre 0 et `RAND_MAX`, suivant une distribution uniforme.

La suite de nombres aléatoires est toujours la même, sauf si on change l'initialisation du générateur avec la fonction `srand(int)`. Pour être sûr de l'aléatoire de cette fonction, on utilise la fonction `time()` de la bibliothèque `<ctime>` pour renvoyer l'heure de l'ordinateur et utiliser, à chaque exécution, une initialisation différente. Il faudra donc la fonction `srand(time(0))`, avec la fonction `time(0)` permettant de prendre l'heure de l'ordinateur.

Dans le cas d'un programme où l'on souhaite avoir en sortie un nombre aléatoire entre 0 et 1, on divise le nombre aléatoire obtenu par `RAND_MAX`. Dans le cas de deux nombres réels, on ajoute (double) devant la fonction `rand()` pour obtenir un nombre aléatoire réel. On peut aussi le faire en affichant le reste de la division par n + 1 pour n'importe quel nombre.