Chapitre 1

Les éléments de base du langage $C++$

$C++$ est l'un des langages de programmation les plus utilisés. Il offre au programmateur des outits puissants pour écrire des programmes efficaces, structurés et orientés objet.

1.1 Structure générale d'un programme $C++$

Un programme $C++$ consiste en plusieurs composants : des **commentaires**, des **directives du préprocesseur**, des **constantes**, des **variables** et des **fonctions globales**, des **classes** et des **objets**.

Voici un exemple d'un programme $C++$:

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

int main()
{
   cout << "Programmation en C++ !!!";
   return 0;
}
```

Le résultat d'exécution de ce programme est :

```
Programmation en C++ !!!
```

1.1.1 Directives du préprocesseur

Une **directive du préprocesseur** est une commande qui commençe par le symbole '#'. Le préprocesseur est un module qui lit le texte source du programme

avant sa compilation et exécute uniquement les lignes commençant par le '#'. Son rôle est de **préparer le code source à la compilation**.

Par exemple, la directive :

```
#include <iostream>
```

a pour effet d'**insérer le contenu d'un fichier nommé** *iostream* **dans le programme initial**. Le mot *iostream* est écrit entre les symboles '<' et '>'. Le fichier **iostream** contient le code qui permet à un programme C++ d'afficher son résultat sur écran. Comme un programme doit utiliser **cout** pour afficher un résultat sur écran, le fichier *iostream* doit être inclus. Le fichier *iostream* est appelé un **fichier d'en-tête**, et doit être inclus en haut (en tête) du programme.

1.1.2 L'organisation des noms dans un programme

L'instruction C++ **using** dans la ligne de code :

```
using namespace std;
```

permet d'**organiser les noms des entités** au sein d'un programme. La ligne précédente signifie que le programme accédera aux entités dont les noms font partie de l'**espace des noms** appelé **std**. La raison pour laquelle le programme a besoin d'accéder à l'espace de noms **std** réside dans le fait que chaque nom crée par le fichier **iostream** est une partie de cet espace de noms.

1.1.3 Déclarations des fonctions et programme principal

En *C* **ANSI**, si on ne déclare pas une fonction avant son utilisation, le compilateur suppose qu'elle retourne un entier (**int**) par défaut. En C++, la règle est stricte :

Toute fonction doit être déclarée ou définie avant sa première utilisation.

En plus, la fonction **main** doit retourner un **int** (0 par défaut) au système d'exploitation après la fin de son exécution. Bien que la plupart des programmes C++ aient plus qu'une fonction, chaque programme C++ doit avoir une fonction appelée **main**(). Elle constitue le point d'entrée du programme.

1.2 Les commentaires

En plus des fameux commentaires /* . . . */ disponible en *C*, le langage C++ offre un autre moyen simple pour écrire un commentaire dit de **fin de ligne** :

```
// Un commentaire C++ qui tient jusqu'à la fin de ligne
```

Le compilateur ignore tout texte écrit dans le code à partir du symbole **double slash** "//" jusqu'à la fin de la ligne.

1.3 Les flots d'entrée et de sortie

Une différence nette entre *C++* et *C* réside dans les **opérations d'entrée/sortie**. Alors que les opérations d'E/S dans *C* sont toutes effectuées via des fonctions de la librairie standard <**stdio.h**> (comme **printf** et **scanf**), ces opérations sont réalisées par deux opérateurs en *C++*, appelés **opérateurs de flots de données**. Il s'agit de l'**opérateur de flot d'entrée** ">>" et l'**opérateur de flot de sortie** "<<".

Un **flot de données** en *C++* représente un **transfert des données** entre la mémoire principale de l'ordinateur et un de ses périphériques. Un **flot d'entrée** est un transfert des données d'un périphérique d'entrée (le clavier par défaut) vers la mémoire centrale. Un **flot de sortie** est un transfert des données de la mémoire centrale vers un périphérique de sortie (l'écran par défaut). La figure suivante illustre le principe de flots de données dans *C++* :

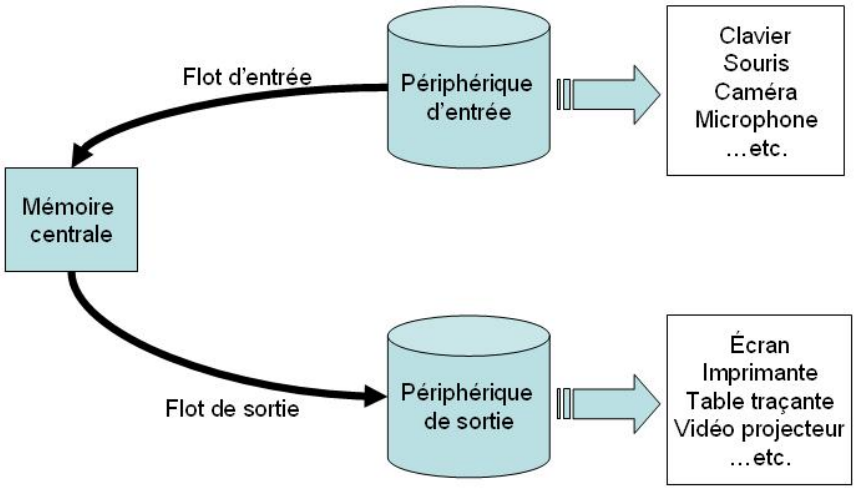


FIG. 1.1 – Le principe de flots d'E/S en *C++*.

1.3.1 L'opérateur de flot de sortie <<

L'opérateur de flot de sortie "<<" permet d'**insérer des valeurs dans le flot de sortie** dont le nom qui apparaît à gauche de l'opérateur. L'**objet cout** est le **flot de sortie standard** et est le plus utilisé en *C++*. Par défaut, cet objet désigne l'écran de l'ordinateur. Par exemple, l'instruction

```
cout << "Programmation en C++ !!!";
```

affichera sur écran le texte : **Programmation en *C++* !!!**.

En général, un opérateur est une chose qui réalise une action sur un ou plusieurs objets. L'action réalisée par l'opérateur "<<" est **l'envoi de la valeur de l'expression indiquée à sa droite au flot de sortie indiquée à sa gauche**. Les valeurs passées à droite seront envoyées selon leur ordre d'apparution. Avec cet opérateur, il est possible d'afficher n'importe quelle valeur de n'importe quel type (*char*, *int*, *float*, *double*, *chaîne de caractères*, ...etc). Voici un exemple :

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

int main()
{
   int a = 107;
   float b = 1.2345;
   char c = 'z';
   double x = 4.25E-19;
   char mess[] = "Bonjour";

   cout << a << ' ' << b << " " << c << "\n";
   cout << x << " " << mess << "\n";
   return 0;
}
```

Dans ce programme, on demande successivement :

1. L'affichage de la valeur de la variable a de type *int*, d'un seul caractère espace, de la valeur de la variable b de type *float*, de la chaîne formée par un seul espace, de la valeur de la variable c de type *char* et d'un seul caractère retour à la ligne.

2. L'affichage de la valeur de la variable x de type *double*, de la chaîne formée par un seul espace, de la valeur de la variable *mess* de type *chaîne de caractères* et d'un seul caractère retour à la ligne.

Le résultat qu'affichera ce programme est :

```
107 1.2345 z
6.78e-009 Bonjour
```

1.3.2 L'opérateur de flot d'entrée >>

L'**objet cin** représente le **flot d'entrée standard** et s'utilise avec l'opérateur de flot d'entrée ">>". Par défaut, l'objet **cin** est associé au clavier. Il permet alors

de saisir des valeurs à partir du clavier. Comme *cout*, l'objet **cin** n'a pas besoin d'un spécificateur de format. Voici un exemple :

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

int main()
{
   int largeur, longueur, surface;

   cout << "Ce programme calcule la surface d'un";
   cout << "rectangle.\n";
   cout << "Donnez la largeur : ";
   cin >> largeur;
   cout << "Donnez la longueur : ";
   cin >> longueur;
   surface = largeur * longueur;
   cout << "La surface est : " << surface << ".\n";
   return 0;
}
```

Il est possible d'utiliser un seul objet **cin** pour effectuer une **lecture mutliple** de données, par exemple :

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

int main()
{
   int largeur, longueur, surface;

   cout << "Ce programme calcule la surface d'un";
   cout << "rectangle.\n";
   cout << "Donnez la largeur et la longueur : ";
   cin >> largeur >> longueur;
   surface = largeur * longueur;
   cout << "La surface est : " << surface << ".\n";
   return 0;
}
```

1.4 Les manipulateurs de flots

1.4.1 Le manipulateur endl

L'**objet endl** (*fin de ligne*) est un ***manipulateur de flot***. Il permet d'insérer un saut de ligne au flot de sortie courant et de vider le tampon de sortie utilisé par l'objet **cout**. Exemple :

```
cout << "Bonjour" << endl;
```

Les expressions :

```
cout << endl;
cout << "\n";
cout << '\n';
```

sont tout à fait équivalentes.

1.4.2 Les manipulateurs dec, oct et hex

Les manipulateurs **dec**, **oct**, et **hex** sont utilisés pour convertir des valeurs dans les trois différente bases : ***décimale***, ***octale*** et ***hexadécimale***. Le manipulateur **dec** est pris par défaut. Les trois s'utilisent avec **cout** et **cin**. Ce premier exemple montre comment utiliser ces manipulateurs pour afficher un entier dans les trois bases habituelles :

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

int main()
{
   int n = 1492; // base 10

   cout << "Base  8: n = " << oct << n << endl;
   cout << "Base 10: n = " << dec << n << endl;
   cout << "Base 16: n = " << hex << n << endl;
   return 0;
}
```

Le résultat affiché par ce programme est :

```
Base  8: n = 2724
Base 10: n = 1492
Base 16: n = 5d4
```

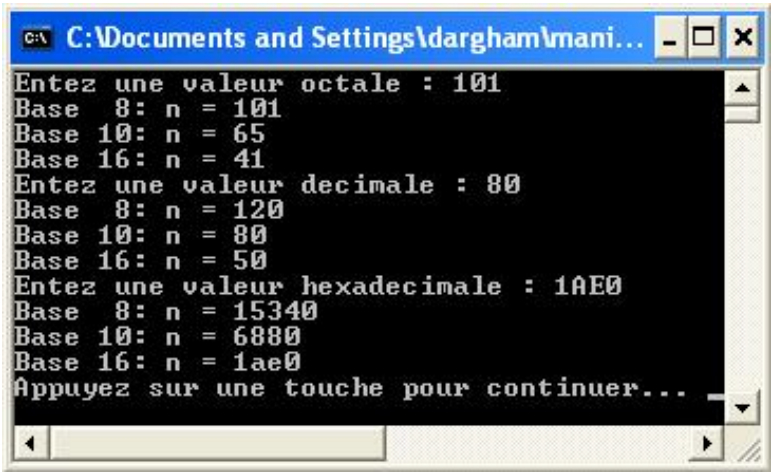
L'exemple suivant montre comment lire la valeur d'un entier en base 8, 10 et 16 :

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

int main()
{
   int n;

   cout << "Entez une valeur octale : ";
   cin >> oct >> n;
   cout << "Base  8: n = " << oct << n << endl;
   cout << "Base 10: n = " << dec << n << endl;
   cout << "Base 16: n = " << hex << n << endl;
   cout << "Entez une valeur decimale : ";
   cin >> dec >> n;
   cout << "Base  8: n = " << oct << n << endl;
   cout << "Base 10: n = " << dec << n << endl;
   cout << "Base 16: n = " << hex << n << endl;
   cout << "Entez une valeur hexadecimale : ";
   cin >> hex >> n;
   cout << "Base  8: n = " << oct << n << endl;
   cout << "Base 10: n = " << dec << n << endl;
   cout << "Base 16: n = " << hex << n << endl;
   system("pause");
   return 0;
}
```

La figure suivante illuste un exemple d'exécution de ce programme :



Si l'un de ces trois manipulateurs est utilisé juste après **cin** ou **cout**, il **persistera**, c'est-à-dire qu'il va être actif jusqu'à l'activation d'un autre manipulateur différent. Voici un exemple :

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

int main()
{
   int n = 11;

   cout << dec << "Base 10: n = " << n << endl;
   cout << "n * 2 = " << n * 2 << endl;
   cout << oct << "Base  8: n = " << n << endl;
   cout << "n * 2 = " << n * 2 << endl;
   cout << hex << "Base 16: n = " << n << endl;
   cout << "n * 2 = " << n * 2 << endl;
   return 0;
}
```

Voici le résultat qui sera affiché par ce programme :

```
Base 10: n = 11
n * 2 = 22
Base  8: n = 13
n * 2 = 26
Base 16: n = b
n * 2 = 16
```

1.4.3 Les manipulateurs setw, setfill, left et right

Le manipulateur de flot **setw**(*int*) s'utilise avec l'objet **cout** et permet de spécifier la **largeur minimale du champ d'affichage**. Par exemple, l'expression **setw**(4) signifie "*mettre la largeur minimale du champ d'affichage à* 4 *colonnes pour la prochaine sortie*". Ce manipulateur est définie dans le fichier <**iomanip**>. Soit k un entier et *exp* une expression à afficher en utilisant l'instruction :

```
cout << setw(k) << exp;
```

et supposant que la valeur de *exp* occupe h colonnes. Alors :

1. si $h < k$, la valeur de *exp* sera **justifiée à droite**.
2. si $h >= k$, la valeur de *exp* sera affichée sur h colonnes.

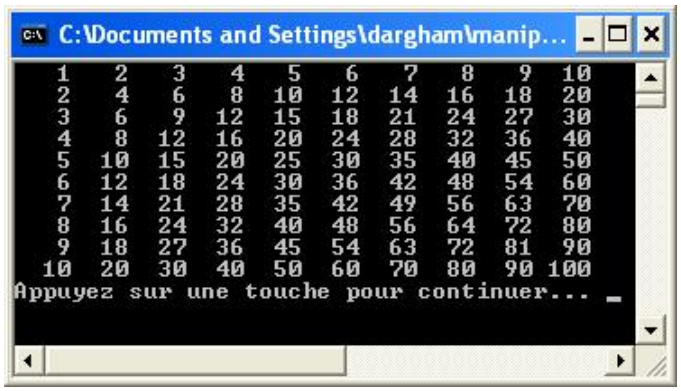
Voici un exemple :

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

int main()
{
   int i, j;

   for(i = 1; i <= 10; i++)
   {
      for(j = 1; j <= 10; j++)
         cout << setw(4) << i * j;
      cout << endl;
   }
   system("pause");
   return 0;
}
```

Le résultat d'exécution de ce programme est le suivant :



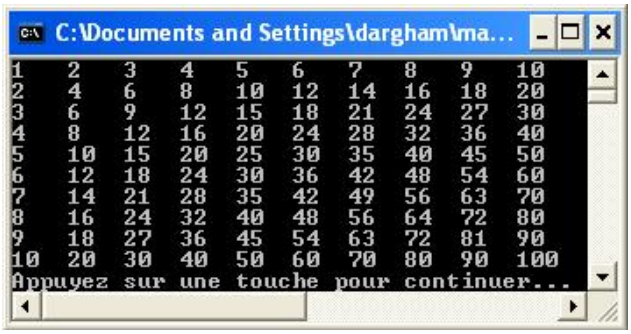
Le manipulateur **left** est utilisé pour forcer l'**alignement à gauche**. En fait, le manipulateur **setw**() utilise l'option **right** par défaut. Attention, **left** et **right** sont **persistants**, par contre **setw**() ne l'est pas. Voici le même exemple que le précédent mais avec un ajustement à gauche :

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

int main()
{
   int i, j;
```

```
   for(i = 1; i <= 10; i++)
   {
      for(j = 1; j <= 10; j++)
         cout << setw(4) << left << i * j;
      cout << endl;
   }
   system("pause");
   return 0;
}
```

Le programme produira le résultat suivant :



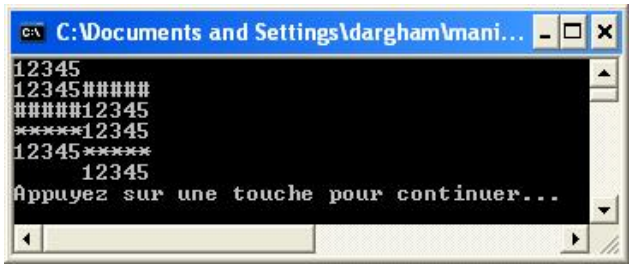
Le manipulateur **setfill**(*char*) s'utilise avec **setw**() et permet de **choisir un caractère de remplissage du vide** dans le cas d'une largeur réelle plus petite que celle spécifiée avec **setw**(). Ce manipulateur est lui aussi **persistant** et prends le **caractère espace comme paramètre par défaut**. Voici un exemple :

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

int main()
{
   int n = 12345;

   cout << setw(10) << left << n << endl;
   cout << setw(10) << setfill('#') << n << endl;
   cout << setw(10) << right << n << endl;
   cout << setw(10) << setfill('*') << n << endl;
   cout << setw(10) << left << n << endl;
   cout << setw(10) << right << setfill(' ') << n << endl;
   system("pause");
   return 0;
}
```

Le résultat affiché est le suivant :



1.4.4 Les manipulateurs setprecision, fixed et showpoint

Il est possible de spécifier le nombre de **chiffres significatifs** des valeurs réelles en utilisant le manipulateur **setprecision**(*int*) avec l'objet **cout**. L'expression "**setprecision**(n)" permet d'afficher un nombre réel en utilisant n chiffres signifiatifs. Le programme suivant montre les résultats d'une opération de division affichés en utilisant ce manipulateur :

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

int main()
{
   double quotient, number1 = 132.364, number2 = 26.91;

   quotient = number1 / number2;
   cout << quotient << endl;
   cout << setprecision(5) << quotient << endl;
   cout << setprecision(4) << quotient << endl;
   cout << setprecision(3) << quotient << endl;
   cout << setprecision(2) << quotient << endl;
   cout << setprecision(1) << quotient << endl;
   return 0;
}
```

Voici le résultat produit par ce programme :

```
4.91877
4.9188
4.919
4.92
4.9
5
```

Par opposition à **setw**(), le manipulateur **setprecision**() est **persistant**. Lorsque la précision d'un nombre est mise à une valeur trop petite, les nombres tendent à être affichés en notation scientifique. Par exemple le programme suivant :

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

int main()
{
   double x = 321.57, y = 269.62, z = 307.77;

   cout << "x = " << setprecision(5) << x << endl;
   cout << "y = " << y << endl;
   cout << "z = " << z << endl;
   x = 145678.99;
   y = 205614.85;
   z = 198645.22;
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   cout << "z = " << z << endl;
   return 0;
}
```

donnera le résultat suivant :

```
x = 321.57
y = 269.62
z = 307.77
x = 1.4568e+005
y = 2.0561e+005
z = 1.9865e+005
```

Le manipulateur **fixed** force **cout** à afficher les chiffres en notation **virgule-fixe** ou **décimale**. Voici un exemple illustrant son utilisation :

```
double x = 123.4567;
cout << setprecision(2) << fixed << x << endl;
```

Comme le manipulateur **fixed** est utilisé, le manipulateur **setprecision**() va causer l'affichage du nombre x avec 2 chiffres après la virgule. La valeur qui sera affiché est alors 123.46 (après arrondi). Notons que le manipulateur **fixed** est **persistant**, tout comme **setprecision**().

Par défaut, les nombres réels qui n'ont pas une partie fractionnaire ne sont pas affichés avec le point décimal. Par exemple, dans le code suivant :

```
double x = 123.4, y = 456.0;
cout << setprecision(6) << x << endl;
cout << y << endl;
```

les instructions **cout** vont afficher :

```
123.4
456
```

Même si l'on a spécifie 6 chiffres significatifs pour les deux nombres, aucun nombre n'est affiché avec des 0 après la virgule. Le manipulateur **showpoint** permet de résoudre ce problème. Voici un exemple :

```
double x = 123.4, y = 456.0;
cout << setprecision(6) << showpoint << x << endl;
cout << y << endl;
```

Ces deux **cout** produiront l'affichage suivant :

```
123.400
456.000
```

1.5 Les constantes

En *C*, la notion de constante n'existe pas. Mais le programmaeur peut définir des *constantes symboliques* en utilisant la directive #**define** du pré-processeur. En revanche, *C++* dispose du mot-clé **const** pour définir des constantes :

```
const int N = 10;
const double pi = 3.14159265353;
const char message[] = "Bonjour";
```

Les entités *N*, *pi* et *message* sont **crées en mémoire** et le **compilateur les protège contre toute opération d'écriture**. Ainsi, les instructions suivantes génèrent des erreurs :

```
N++;
cin >> pi;
message[0] = 'b';
```

1.6 Déclarations de variables

En *C*, la règle veut que toutes les déclarations doivent être effectuées avant la première instruction exécutable du programme. Cette limitation n'est plus valable en *C++* : le programmeur peut déclarer une variable là où il veut et au momment qu'il décide.

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

int main()
{
   for(int i = 0; i <= 10; i++)
   {
      int carre = i * i;
      cout << i << " * " << i << " = ";
      cout << carre << endl;
      int cube = carre * i;
      cout << "\t" << i << " ^ 3 = ";
      cout << cube << endl;
   }
   cout << i << endl;
   // Erreur : i non reconnu ici!
   return 0;
}
```

1.7 Le type de données bool

Une **variable booléenne** ne peut avoir que deux valeurs : **true** (*vrai*) ou **false** (*faux*). Il n'existe pas de type **booléen** en *C*. Cependant, *C++* dispose d'un type de base qui permet de représenter des valeurs booléennes. Il s'agit du type nommé **bool** (*bool* est désormais un mot-clé du langage *C++*). Voici un exemple d'utilisation de ce type :

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

int main()
{
   bool boolValue;

   boolValue = true;
   cout << boolValue << endl;
```

```
   boolValue = false;
   cout << boolValue << endl;
   return 0;
}
```

Ce programme affiche :

```
1
0
```

En effet, la valeur booléenne **true** est représentée par la valeur 1 et la valeur booléenne **false** par la valeur 0, comme dans *C*. Le type **bool** est tout simplement un sous-type du type **int**.

1.8 La surcharge de fonctions

Surcharger une fonction c'est lui attribuer plusieurs définitions dans le même programme. Ceci est impossible en *C*, mais possible ne *C++*. Exemple :

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

void print(int a)
{
   cout << "un entier : " << a << endl;
}

// surcharge de la fonction print

void print(double a)
{
   cout << "un double : " << a << endl;
}

int main()
{
   print(17);
   print(17.25);
   return 0;
}
```

Voici le résultat de l'exécution de ce programme :

```
un entier : 17
un double : 17.25
```

1.9 Les arguments par défaut

L'argment d'une fonction $C++$ peut avoir une **valeur par défaut**. Cependant, seuls les derniers arguments de la droite vers la gauche peuvent être avec des valeurs par défaut. Exemple :

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

void init(int, int = 3);  // déclaration

int main()
{
   init(1, 7); // affiche : 1 et 7
   init(1);    // affiche : 1 et 3
   return 0;
}

// Définition de la fonction init
void init(int a, int b)
{
   cout << a << " et " << b << endl;
}
```

1.10 Les fonctions en ligne

L'appel d'une fonction ordinaire C ou $C++$ est souvent coûteux. En effet, un temps et un espace mémoire supplémentaires sont requises pour invoker l'appel d'une function, y passer ses paramètres, allouer la mémoire pour ses variables locales, sauver les variables courantes et l'endroit d'exécution dans le programme principal, ...etc. Dans certains cas, il est très util d'éviter tout ça en spécifiant que la fonction doit être **en ligne**. Cela demande au compilateur de **remplacer chaque appel de la fonction par le code explicite de celle-ci**.

Pour spécifier qu'une fonction est **en ligne**, on précède sa déclaration par le mot-clé **inline**. Exemple :

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

inline int cube(int x)
{
   return x * x * x;
```

```
}

int main()
{
   cout << cube(4) << endl;
   int x, y;
   cin >> x;
   y = cube(2 * x + 3);
   return 0;
}
```

La fonction **main** va être remplacée par le code suivant :

```
int main()
{
   cout << (4) * (4) * (4) << endl;
   int x, y;
   cin >> x;
   y = (2 * x + 3) * (2 * x + 3) * (2 * x + 3);
}
```

1.11 Le passage d'arguments par référence

1.11.1 Le passage par valeur

En *C*, l'unique mode de passage des paramètres est le **passage par valeur**. Cela signifie que l'expression transmise lors de l'appel de la fonction est évaluée en premier lieu, et ensuite sa valeur est assignée au paramètre correspondant dans la liste des paramètres de la fonction avant l'exécution du code de celle-ci. Ce mécanisme permet d'utiliser des expressions plus compliquées à la place d'un argument formel dans l'appel d'une fonction. Par exemple la fonction cube() peut également être appelée comme cube(3), cube($2 * x + 3$), ou même qoomme cube($2 * sqrt(x) - cube(3)$). Dans chacun des cas précédents, l'expression entre parenthèses est transformée en une valeur unique qui est ensuite passée à la fonction.

Cependant, il y a des situations où la fonction a besoin de **modifier la valeur de l'un de ses paramètres**. Le passage par valeur ne permet pas de résoudre ce problème. Par exemple, soit la fonction suivante qui multiplie par 2 son unique argument :

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

```
void multiplier_par_2(int x)
{
   x = 2 * x;
}

int main()
{
   int x = 7;
   cout << "La valeur de x avant l'appel est : ";
   cout << x << endl;
   multiplier_par_2(x);
   cout << "La valeur de x apres l'appel est : ";
   cout << x << endl;
   return 0;
}
```

Le résultat produit par le programme précédent est comme suivant :

```
La valeur de x avant l'appel est : 7
La valeur de x apres l'appel est : 7
```

La fonction n'a pas modifiée le paramètre x, car le passage par valeur ne le favorise pas.

1.11.2 Le passage par adresse

Une solution en C du problème cité ci-dessus est d'utiliser les **pointeurs**. Voici le code à écrire :

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

void multiplier_par_2(int *x)
{
   *x = 2 * (*x);
}

int main()
{
   int x = 7;
   cout << "La valeur de x avant l'appel est : ";
   cout << x << endl;
   multiplier_par_2(x);
   cout << "La valeur de x apres l'appel est : ";
```

```
   cout << x << endl;
   return 0;
}
```

Le mode de passage est le même (même s'il est appelé dans ce cas **passage par adresse**), mais comme le paramètre x de la fonction est un pointeur, sa valeur est une adresse. Notons que dans ce cas, il faut donner une adresse à la fonciton, comme dans l'exemple :

```
multiplier_par_2(&x);
```

Le résultat produit par cette deuxième version du programme est le suivant :

```
La valeur de x avant l'appel est : 7
La valeur de x apres l'appel est : 14
```

Puisque la valeur passée est l'adresse de la variable x, la fonction utilisera cette valeur pour exécuter son code. Par ce moyen, la fonciton arrive finalement à changer la valeur de la variable x.

1.11.3 Le passage par référence

$C++$ propose une solution élégante pour ce même problème : le **passage par référence**. Pour passer un paramètre en référence, il faut suffixer le type de ce paramètre par le symbole &. Cela a pour effet de rendre la variable locale une référence à l'argument passé à la fonction. Toute modification de la variable locale à l'intérieur de la fonction causera une modification identique à l'argument qui sera passée à la fonction. Voici le code à écrire en $C++$:

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

void multiplier_par_2(int &x)
{
   x = 2 * x;
}

int main()
{
   int x = 7;
   cout << "La valeur de x avant l'appel est : ";
   cout << x << endl;
   multiplier_par_2(x);
   cout << "La valeur de x apres l'appel est : ";
```

```
   cout << x << endl;
   return 0;
}
```

Le résultat de ce code est :

```
La valeur de x avant l'appel est : 7
La valeur de x apres l'appel est : 14
```

Notons deux points importants ici : la simplicité du code et l'obtention du résultat volu.

1.12 La portée d'un identificateur

La **portée** d'un identificateur désigne l'endroit du programme où cet identificateur est accessible (c'est-à-dire où l'identificateur peut être utilisé). Cet endroit commençe à partir de la ligne où l'identificateur a été déclaré. Si cette déclaration est faite à l'intérieur d'une fonction (la fontion main() étant incluse), alors la porté s'étende jusqu'à la fin du bloc **le plus profond qui contient la déclaration**. Un programme *C* ou *C++* peut avoir plusieurs objets ayant le même nom **si leurs portées sont imbriquées ou disjointes**. Cela est illustré par l'exemple suivant :

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

void f();    // la fonction f() est globale
void g();    // la fonction g() est globale
int x = 11;  // la variable x est globale

int main()
{
   int x = 22;
   {
      int x = 33;
      cout << "In block inside main(): x = ";
      cout << x << endl;
   }  // fin de la portée du bloc interne
   cout << "In main(): x = " << x << endl;
   cout << "In main(): ::x = ";
   cout << ::x << endl;
   // accès à la variable globale x
   f();
```

```
   g();
} // fin de la portée du main

void f()
{
   int x = 44;
   cout << "In f(): x = " << x << endl;
} // fin de la portée de f()

void g()
{
   cout << "In g(): x = " << x << endl;
} // fin de la portée de g()
```

Dans cet exemple, f() et g() sont deux fonctions globales, et le premier x est une variable globale. Par suite, leur portée inclue le fichier tout entier. Le second x est déclaré à l'intéruer du main(), il a donc une portée locale, c'est-à-dire il est uniquement accessible à l'intérieur de la fonction main(). Le troisième nom x est déclaré à l'intérieur d'un bloc interne, et donc sa portée est restreinte à ce bloc interne.

L'**opérateur de résolution de portée** notée :: est utilisé pour accéder au dernier nom x dont la portée a été réceémment écrasée. Dans notre exemple, :: x désigne la variable globale x dont la valeur est 11. La variable x initializée avec 44 a une portée limitée à la fonction f(). L'affichage produit par le programme de l'exemple est :

```
In block inside main(): x = 33
In main(): x = 22
In main(): ::x = 11
In f(): x = 44
In g(): x = 11
```

1.13 L'allocation dynamique de la mémoire

Jusqu'à présent, nous avons vu que des situations où le nombre de variables pour l'exécution d'un programme se fait à l'avance pendant la **phase statique** (c'est-à-dire, avant la compilation du programme). Par exemple, un programme qui va calculer la surface d'un rectangle nécessitera 3 variables : une pour la largeur du rectangle, une pour sa longueur, et une pour stocker la surface. De même, si l'on veut écrire un programme qui calcule les salaires de 30 employés, on aura probablement besoin de créer un tableau de 30 éléments pour sauver le salaire

pour chaque personne.

Cependant, il y a des situations où le nombre de variables à définir dans le programme n'est pas connue à l'avance (car ce nombre est lui-aussi un paramètre du programme). Par exemple, supposons que nous voulons écrire un programme qui calculera la moyenne des notes d'une classe dont le nombre des élèves sera saisi pendant l'exécution du programme. La solution à ce problème consiste à donner à un programme la possibilité de créer les variables "**à la volée**". Cette technique s'appelle l'**allocation dynamique de la mémoire** et est possible uniquement via l'utilisation des **pointeurs** (aussi bien en *C* qu'en *C++*).

L'allocation dynamique de la mémoire signifie qu'un programme en cours d'exécution formule une demande à l'ordinateur de lui **réserver un espace mémoire libre** de **taille assez suffisante** pour héberger la valeur d'une variable d'un type de données spécifique. Pour satisfaire cette demande, l'ordinateur cherchera une zone mémoire libre (non utilisée par un autre programme) de taille adéquate, et en cas de réuissite, il retournera au programme l'**adresse de début** de cette zone mémoire. Le programme peut accéder à cette zone mémoire qui est récémment allouée uniquement via son adresse, d'où la nécessité d'utiliser un **pointeur**.

En *C*, l'allocation dynamique de la mémoire se fait par l'emploi des 4 fonctions célèbres de la bibliothèque <**stdlib.h**> : **malloc**(), **calloc**(), **realloc**() et **free**(). En *C++*, l'allocation dynamique de la mémoire est simplifiée par l'utilisation de deux opérateurs : **new** et **delete**. L'opérateur **new** permet d'allouer dynamiquement un espace mémoire pour une variable simple ou un tableau, alors que l'opérateur **delete** libère un espace mémoire déjà allouée via *new*. Voici un exemple :

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

int main()
{
   int *iptr;
   int *tabptr;

   iptr = new int;
   // alloue de la mémoire pour un entier
   *iptr = 25;
   cout << *iptr << endl; // Affiche 25
   cin >> *iptr;
   /* Lit une valeur et le stocke dans
      la zone pointée par iptr */
```

```
   delete iptr;
   // libère l'espace mémoire allouée par iptr

   tabptr = new int[100];
   /* alloue de la mémoire pour
      un tableau de 100 entiers */
   for(int i = 0; i < 100; i++)
      tabptr[i] = i * i;
   delete [] tabptr;
   /* libère l'espace mémoire allouée
      pour le tableau */
   return 0;
}
```

1.14 Les structures en $C++$

1.14.1 Déclaration de structures

Comme C, le langage $C++$ permet de *grouper plusieurs variables* de **types et de noms différents** au sein d'une entité unique appelée une **structure**. Ces variables sont normalement reliées par une relation logique. Par exemple, un système de paie a besoin de garder, pour chaque employé, des variables comme :

- le code de l'employé.
- le nom de l'employé.
- le nombre d'heures de travail qu'a effectué l'employé.
- le taux horaire de l'employé.
- le salaire de l'employé.

Toutes ces variables individuelles sont logiquement reliées, car leur ensemble permet de représenter les informations sur un même employé. Pour les relier toutes, on les regrouper dans une unique structure. Voici comment faire ceci en $C++$:

```
struct PaieInfo
{
   int empCode;      // code de l'employé
   char nom[55];     // nom de l'employé
   double heurs;     // heures de travail de l'employé
   double tauxHeure; // taux horaire de l'employé
   double salaire;   // le salaire de l'employé
};
```

Les variables regroupées au sein d'une même structure s'appellent les **données membres** de cette structure. La déclaration précédente a permet de créer un

nouveau type de données nommé *PaieInfo*. Il est alors possible de déclarer une variable de ce type. Ainsi, on peut effectuer les déclarations suivantes :

```
PaieInfo employe;
PaieInfo lesEmployes[100];
PaieInfo *ptrEmploye;
```

Dans la première instruction, on a déclaré une variable structurée nommée ***employe*** de type **PaieInfo**. Dans le la deuxième instruction, on a déclaré un tableau nommé ***LesEmployes*** de 100 éléments de type **PaieInfo**. Dans la troisième instruction, on a déclaré un pointeur nommé ***ptrEmploye*** vers un objet de type **PaieInfo**.

1.14.2 Accèes à une donnée membre d'une structure

L'opérateur point "." permet d'accéder à une **donnée membre individuelle** d'une structure. Voici un exemple :

```
PaieInfo employe;

employe.empCode = 1000;
cin >> employe.nom;
employe.heurs = 39;
employe.tauxHeure = 150;
employe.salaire = employe.heurs * employe.tauxHeure;
cout << "Salaire = " << employe.salaire << endl;
```

Le code suivant permet d'appliquer une augmentation de 2% pour tous 100 employés d'une société :

```
PaieInfo lesEmployes[100];

for(int i = 0; i < 100; i++)
   lesEmployes[i].salaire *= 1.02;
```

Le code suivant permet de créer un employé dynamiquement et de saisir ses informations :

```
PaieInfo *ptrEmploye;

ptrEmploye = new PaieInfo;
cout << "Code employe : ";
cin >> ptrEmploye->empCode;
cout << "Nom employe : ";
```

```
cin >> ptrEmploye->nom;
cout << "Nombre d'heures : ";
cin >> ptrEmploye->heurs;
cout << "Taux horiare : ";
cin >> ptrEmploye->tauxHeure;
ptrEmploye->salaire = ptrEmploye->heurs
                              *
                      ptrEmploye->tauxHeure;
cout << "Salaire = " << ptrEmploye->salaire << endl;
```

Le symbole "$->$" permet d'accéder au donnée membre d'une structure pointée par un pointeur. Ainsi, l'expression :

```
ptrEmploye->empCode
```

est équivalente à :

```
(*ptrEmploye).empCode;
```

1.14.3 Initialisation d'une structure

Il est possible d'initialiser une variable structurée au moment de sa déclaration. Voici un exemple :

```
struct personne
{
   char nom[256];
   char prenom[256];
   int age;
};
personne pers1 = {"Ashille", "Kroneker", 48};
personne tabPers[2] = {
                       {"Ashton", "Flicker", 50},
                       {"Ashille", "Kroneker", 48}
                      };
```

1.14.4 Affectation d'une structure à une autre

Il est possible d'affecter globalment une variable structurée y à une autre x. Cela a pour effet une **recopie membre à membre** de toutes les données membres de y vers x : $y.membre$ sera affecté à $x.membre$. Voici un exemple :

```
struct personne
{
   char nom[256];
   char prenom[256];
   int age;
};
personne pers1 = {"Ashille", "Kroneker", 48};
personne pers2 = pers1; // recopie membre à membre
```

1.14.5 Imbrication de structures

Il est possible qu'une donnée membre d'une structure soit elle-même une structure d'un autre type. Voici un exemple :

```
struct personne
{
   char nom[256];
   char prenom[256];
   int age;
};

struct employe
{
   personne persInfo; // infos personnelles
   int empCode;
   double heurs;
   double tauxHeure;
   double salaire;
};
```

La donnée membre ***persInfo*** est de type ***personne***, c'est-à-dire une structure. Pour accéder au ***nom*** d'un employé, il faut passer par la structure ***persInfo***. Voici un exemple :

```
employe empl;
cout << "Nom de l'employe : ";
cout << empl.persInfo.nom << endl;
```

1.14.6 Structures et fonctions

Comme *C*, une fonction *C++* peut avoir comme argument : une structure, un tableau de structures, un pointeur vers une structure ou une **référence constante** à une structure. L'exemple suivant illustre ces 4 cas de figures :

```
struct Rectangle
{
   double length; // longueur
   double width;  // largueur
   double area;   // surface
};

void showRect(Rectangle r)
{
   cout << r.length << endl;
   cout << r.width << endl;
   cout << r.area << endl;
}

double sumAreas(Rectangle r[], int n)
{
   double t = 0.0;
   for(int i = 0; i < n; i++)
      t += r[i].area;
   return t;
}

double circomference(Rectangle *r)
{
   return 2.0 * (r->length + r->width);
}

double getArea(const Rectangle &r)
{
   return r.area;
}
```

La fonction **showRect**(*Rectangle r*) a comme argument une structure r de type *Rectangle*. Lors de l'appel de cette fonction, les données membres du paramètre passé seront recopiées membre à membre.

La fonction **sumAreas**(*Rectangle r*[ ], *int n*) a comme argument un tableau r de structures de type *Rectangle*. L'argument n désigne le nombre des éléments du tableau r.

La fonction **circomference**(*Rectangle *r*) a comme argument un pointeur r vers une structure de type *Rectangle*. Lors de l'appel de cette fonction, seule l'adresse du paramètre passé est recopiée : il n'y a pas recopie de structures, mais il peut

y avoir une modification de la structure pointée par l'argument passé.

La dernière fonction **getArea**(**const** *Rectangle* &*r*) a comme argument une référence constante *r* à une structure de type *Rectangle*. Lors de l'appel de cette fonction, seule l'adresse du paramètre passé est recopiée : il n'y a pas recopie de structures, de plus l'argument passé est protégé contre toute écriture.

De même, une fonction *C++* peut retourner une structure, un pointeur vers une structure ou une référence à une structure. Voici un exemple de trois fonctions qui recopient une structure *Rectangle* donnée en argument :

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

struct Rectangle
{
   double length;
   double width;
   double area;
};

void showRect(Rectangle r)
{
   cout << "length = " << r.length << endl;
   cout << "width  = " << r.width << endl;
   cout << "area   = " << r.area << endl;
   cout << endl;
}

// la première fonction renvoie une struture
Rectangle copyRect1(Rectangle r)
{
   Rectangle cr;
   cr = r;
   return cr;
}

/* la deuxième fonction renvoie un
   pointeur vers une struture */
Rectangle *copyRect2(Rectangle r)
{
   Rectangle *cr;
   cr = new Rectangle;
```

```
   *cr = r;
   return cr;
}

/* la troisième fonction renvoie une
   référence à une struture */
Rectangle& copyRect3(Rectangle r)
{
   Rectangle &cr = r;
   return cr;
}

int main()
{
   Rectangle r = {4.0, 5.0};

   r.area = r.length * r.width;
   showRect(r);

   Rectangle r1 = copyRect1(r);
   showRect(r1);

   Rectangle *r2 = copyRect2(r);
   showRect(*r2);

   Rectangle &r3 = copyRect3(r);
   showRect(r3);

   system("pause");
}
```