République Algérienne Démocratique et Populaire

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

Université d'Oran des Sciences et de la Technologie Mohamed Boudiaf USTO-MB

Faculté de Génie Electrique

Département d'électronique

Polycopié de Cours

Programmation Orientée Objets en C++

Présenté par :

Dr. MEDDEBER Lila ***Dr. ZOUAGUI Tarik***

Année Universitaire
2017 - 2018

Plan Pédagogique du cours

Matière : Programmation Orientée Objet en C++

Filière : Electronique, Télécommunication, Génie Biomédical

Niveau : 1ere année Master (ESE, I, RT, IB)[1]

Volume Horaire : 45h Cours + Travaux Pratiques

Coefficient : 2

Crédits : 3

Evaluation : Contrôle continu : 40% ; Examen : 60%.

Objectif général du cours:

L'objectif général de ce cours est de permettre aux étudiants d'aborder les fondements de base de la programmation orientée objets ainsi que la maitrise des techniques de conception des programmes avancés en langage C++.

Les principaux points traités sont:

- Les structures de base du langage C++.
- L'allocation dynamique et la maitrise du fonctionnement des pointeurs.
- Le concept de classes et d'objets, les membres, fonctions membres, fonctions amies et le cas particulier très important des constructeurs et du destructeur.
- La notion d'héritage, simple puis multiple avec les notions de polymorphisme.
- La gestion des exceptions.

[1] ESE : Electronique pour les Systèmes Embarqués, I : Instrumentation, RT : Réseaux et télécommunications, IB : Instrumentation Biomédicale

SOMMAIRE

Chapitre I: Introduction à la Programmation Orientée Objets

I.1 Introduction

La conception par objet trouve ses fondements dans une réflexion menée autour de la vie du logiciel. D'une part, le développement de logiciels de plus en plus importants nécessite l'utilisation de règles permettant d'assurer une certaine qualité de réalisation. D'autre part, la réalisation même de logiciel composée de plusieurs phases, dont le développement ne constitue que la première partie. Elle est suivie dans la majorité des cas d'une phase dite de maintenance qui consiste à corriger le logiciel et à le faire évoluer. On estime que cette dernière phase représente 70 % du coût total d'un logiciel, ce qui exige plus encore que la phase de développement doit produire du logiciel de qualité.
La conception objet est issue des réflexions effectuées autour de cette qualité. Celle-ci peut-être atteinte à travers certains critères [6]:

- **La validité**: c'est-à-dire le fait qu'un logiciel effectue exactement les tâches pour lesquelles il a été conçu.
- **Extensibilité**: C'est-à-dire, la capacité à intégrer facilement de nouvelles spécifications (demandées par les utilisateurs ou imposées par un événement extérieur).
- **Réutilisabilité**: Les logiciels écrits doivent pouvoir être réutilisables, complètement ou en partie. Ceci impose lors de la conception une attention particulière à l'organisation du logiciel et à la définition de ses composantes.
- **Robustesse**: c'est-à-dire l'aptitude d'un logiciel à fonctionner même dans des conditions anormales.

I.2 Modularité

Les critères énoncés au paragraphe précédent influent sur la façon de concevoir un logiciel, et en particulier sur l'architecture logicielle. En effet, beaucoup de ces critères ne sont pas respectés lorsque l'architecture d'un logiciel est obscure. Dans ces conditions, le moindre changement de spécification peut avoir des répercutions très importantes sur le logiciel, imposant une lourde charge de travail pour effectuer les mises à jour.
On adopte généralement une architecture assez flexible pour parer à ce genre de problèmes, basée sur les modules. Ceux-ci sont des entités indépendantes intégrées dans une architecture pour produire un logiciel.

I.3 De la programmation classique vers la programmation orientée objet

Les premiers programmes informatiques étaient généralement constitués d'une suite d'instructions s'exécutant de façon linéaire (l'exécution commence de la première instruction du fichier source et se poursuivait ligne après ligne jusqu'à la dernière instruction du programme).

Cette approche, bien que simple à mettre en œuvre, a très rapidement montré ses limites. En effet, les programmes monolithiques de ce type:

- ne se prêtent guère à l'écriture de grosses applications
- et ne favorisent absolument pas la réutilisation du code.

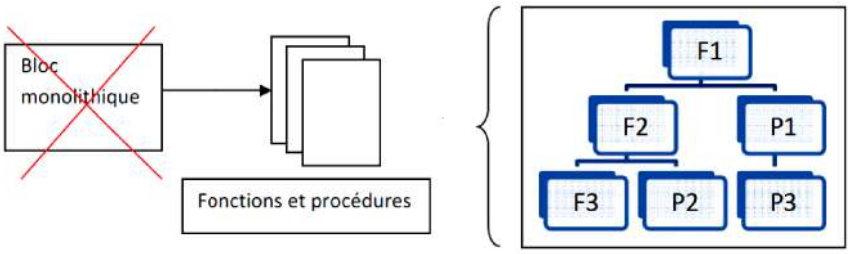
En conséquence, est apparue une autre approche radicalement différente: l'approche procédurale.

L'approche procédurale (classique) consiste à découper un programme en un ensemble de fonctions (ou procédures). Ces fonctions contiennent un certain nombre d'instructions qui ont pour but de réaliser un traitement particulier.

Exemples de traitements qui peuvent être symbolisés par des fonctions:

- Le calcul de la circonférence d'un cercle.
- L'impression d'un relevé de notes d'un étudiant.
- etc.

Dans le cas de l'approche procédurale, un programme correspond à l'assemblage de plusieurs fonctions qui s'appellent entre elles.



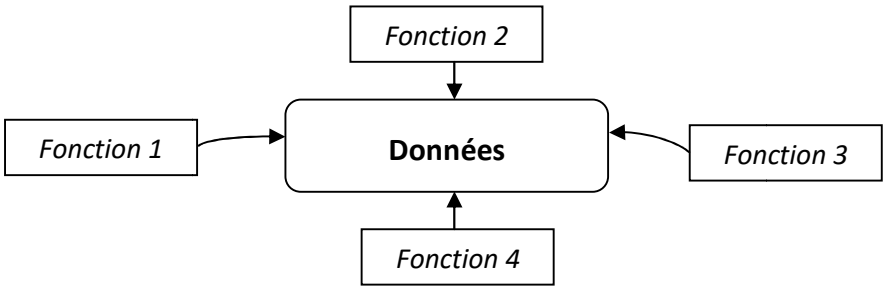
Exemple de langages de programmation procédurale: C, Pascal, Fortran, etc. [13]

L'approche procédurale favorise:

- La création d'un code plus modulaire et structuré.
- La possibilité de réutiliser le même code à différents emplacements dans le programme sans avoir à le retaper.

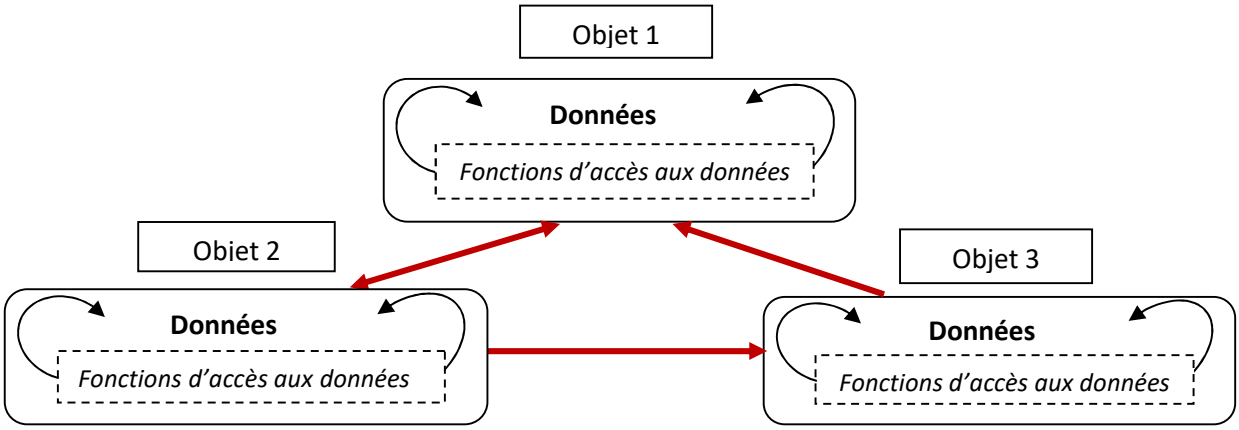
Malgré ses avantages, l'approche procédurale présente également des inconvénients:

- Les fonctions, procédures accèdent à une zone où sont stockées les données. Il y a donc une dissociation entre les données et les fonctions ce qui pose des difficultés lorsque l'on désire changer les structures de données.



- Dans les langages procéduraux, les procédures s'appellent entre elles et peuvent donc agir sur les mêmes données. Il ya donc un risque de partage de données (écriture en même temps dans le même fichier).

De ces problèmes est issus une autre manière de programmer c'est la programmation par objet ou bien *L'approche orientée objet (Début des années 80)*. Selon cette approche, un programme est vu comme un ensemble d'entités (ou objets). Au cours de son exécution, ces entités collaborent en s'envoyant des messages dans un but commun [13].



Nous avons dans ce schéma un lien fort entre les données et les fonctions qui y accèdent.
Mais qu'appelle-t-on un objet ? Que représente un objet ?

I.4 Conceptions par objets

Dans la conception basée sur les données, une réflexion autour des données conduit à :

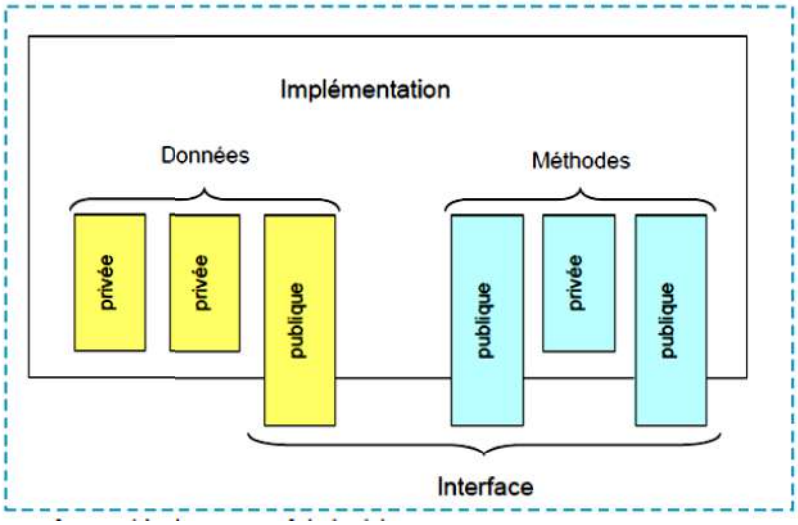
- Déterminer les données à manipuler.
- Réaliser, pour chaque type de données, les fonctions qui permettent de les manipuler.

On parle alors d'**OBJETS**
Un objet est une association de données et des fonctions (méthodes) opérant sur ces données.

Objet = Données + Méthodes

I.4.1 Concepts fondamentaux des objets [6]

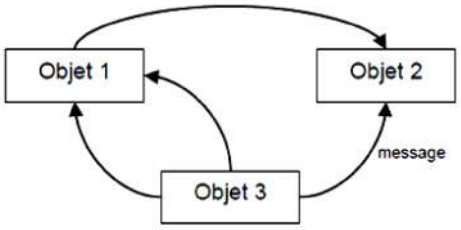
- ***Encapsulation des données*** : consiste à faire une distinction entre l'interface de l'objet et son implémentation.



⇒ Interface : décrit ce que fait l'objet.
⇒ Implémentation : définit comment réaliser l'interface.

Le principe de l'encapsulation est qu'on ne peut agir que sur les propriétés publiques d'un objet: les données sont toutes privées, leur manipulation se fait à travers les méthodes publiques.

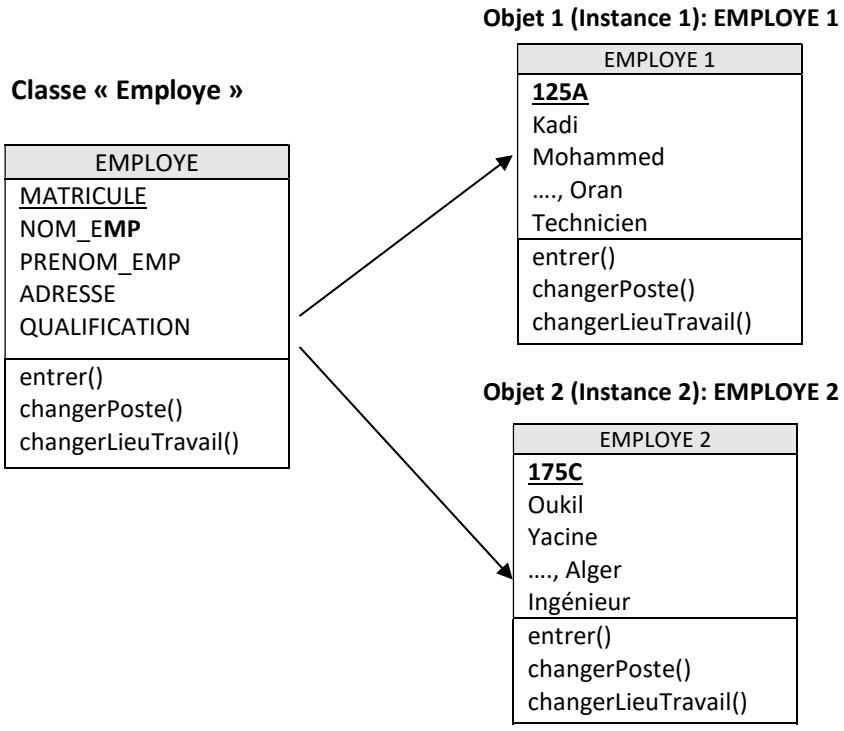
- ***Communication par messages :*** Les objets communiquent entre eux par des messages (un objet demande à un autre objet un service).



- ***Identité et classification***: consiste à regrouper les objets ayant le même comportement pour former un même ensemble, appelé **CLASSE (**cette notion n'est autre que la généralisation de la notion de *type*).

  Un objet d'une classe s'appelle **INSTANCE** de cette classe.

***Exemple*:**



« EMPLOYE 1 » et « EMPLOYE 2 » sont caractérisés par les mêmes propriétés (matricule, nom, prénom, qualification) mais associés à des valeurs différentes. Ils ont le même comportement (entrer/ changerposte,…) mais ont des identités différentes. Et il en serait de même pour tous les employés.

⇒ Tous les employés obéissent à un même schéma

- ***Héritage***: consiste à définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute des nouvelles propriétés (données ou méthodes).

- ***Polymorphisme***: possibilité à divers objets de classes dérivées d'une même classe de répondre au même message. Autrement dit, un même nom peut designer des propriétés de classes différentes.
- ***Généricité***: consiste à définir des classes paramétrées. Une classe générique n'est pas directement utilisable, mais permet de créer des classes dérivées qui peuvent être manipulées.

- ***Modularisation***: Les modules sont construits autour des classes. Un module contiendra l'implémentation d'une classe ou d'un ensemble de classes liées.

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

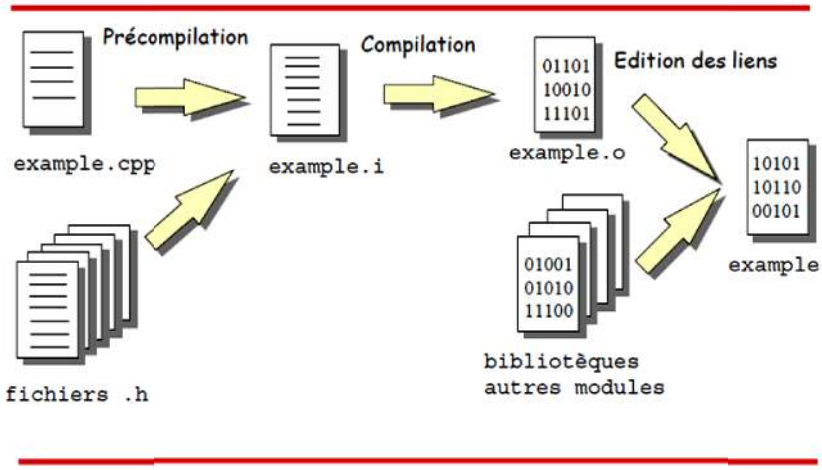
Un suffixe est utilisé pour déterminer le type de fichier

- .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

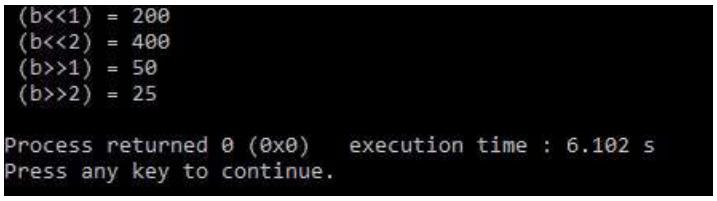
Opérateur	Signification	Arité	Associativité
<<	Décalage à droite	Binaire	Gauche à droite
>>	Décalage à gauche	binaire	Gauche à droite

- 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.

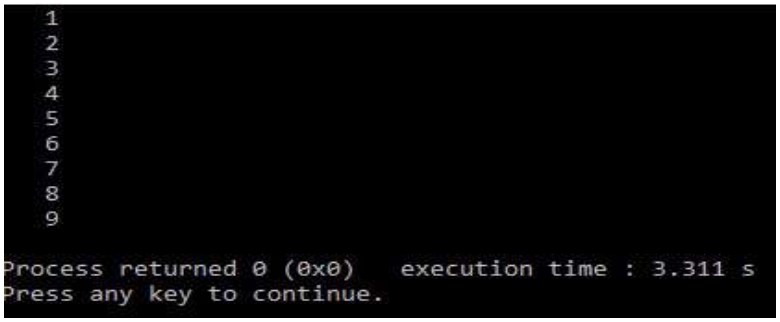
Syntaxe:

```
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.
```

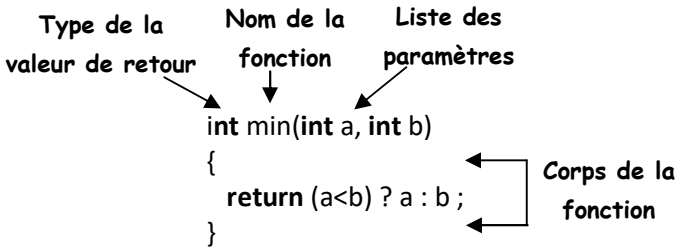
Chapitre III : Fonctions en C++

III.1 Création et utilisation d'une fonction

Une fonction est une opération définie par le programmeur caractérisée par :

- Son nom,
- le type de valeur qu'elle renvoie,
- les paramètres qu'elle reçoit pour faire son travail,
- l'instruction-bloc qui effectue le travail (corps de la fonction).

Exemple:



III. 1.1 Déclaration d'une fonction

Toute fonction doit être déclarée avant d'être appelée pour la première fois. La définition d'une fonction peut faire office de déclaration.

Il peut se trouver des situations où une fonction doit être appelée dans une autre fonction définie avant elle. Comme cette fonction n'est pas définie au moment de l'appel, elle doit être déclarée.

Le rôle des déclarations est donc de signaler l'existence des fonctions aux compilateurs afin de les utiliser, tout en reportant leur définition de ces fonctions plus loin ou dans un autre fichier.

Syntaxe :

```
type nomDeLafonction (paramètres) ;
```

Exemple [4b]***:***

```
double Moyenne(double x, double y);  // Fonction avec paramètres et type retour
char LireCaractere();  //  Fonction avec type de retour et sans paramètres
void AfficherValeurs(int nombre, double valeur); //Fonction avec paramètres et sans type de retour
```

III.1.2 Définition d'une fonction

Toute fonction doit être définie avant d'être utilisée.

Syntaxe:

```
type nomDeLafonction (paramètres)
{
  Instructions ;
}
```

✍ Il est possible de créer *plusieurs fonctions ayant le même nom.* Il faut alors que la liste des arguments des deux fonctions soit différente. C'est ce qu'on appelle la **surcharge** d'une fonction.

Exemple:

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

double moyenne(double a, double b);  // déclaration de la fonction somme :

int main()
{   double x, y, resultat;
    cout << "Tapez la valeur de x : "; cin >> x;
    cout << "Tapez la valeur de y : "; cin >> y;
    resultat = moyenne(x, y);  //appel de notre fonction somme
    cout <<" Moyenne entre "<<  x <<" et "  << y << " est : " << resultat << endl;
 return 0;
}
// définition de la fonction moyenne :
 double moyenne(double a, double b)
{ double moy;
   moy = (a + b)/2;
  return moy;
}
```

- Dans ce programme, on a créé une fonction nommée moyenne qui reçoit deux nombres réels a et b en paramètre et qui, une fois qu'elle a terminé, renvoie un autre nombre moy réel qui représente la moyenne de a et b. l'appel de cette fonction se fait au niveau du programme principale (main).

- Dans cet exemple, le prototype est nécessaire, car la fonction est définie après la fonction main() qui l'utilise. Si le prototype est omis, le compilateur signale une erreur.

- Le prototype peut être encore écrit de la manière suivante:

  double moyenne(**double** , **double**)

Affichage de l'exécution:

```
Tapez la valeur de x : 12.5
Tapez la valeur de y : 5.78
 Moyenne entre 12.5 et 5.78 est : 9.14

Process returned 0 (0x0)   execution time : 34.594 s
Press any key to continue.
```

III.1.3 Fonctions sans retour

C++ permet de créer des fonctions qui ne renvoient aucun résultat. Mais, quand on la déclare, il faut quand même indiquer un type. On utilise le type **void**. Cela veut tout dire : il n'y a vraiment rien qui soit renvoyé par la fonction.

Exemple:

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

void presenterPgm () ;

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

void presenterPgm ()
{
  cout << "ce pgm …";
}
```

✍ Une fonction produit toujours au maximum un résultat, c'est-à-dire qu'Il n'est pas possible de renvoyer plus qu'une seule valeur.

III.2 Surcharge des fonctions [4b]

En C++, Plusieurs fonctions peuvent **porter le même nom** si leurs **signatures diffèrent**. La **signature** d'une fonction correspond aux **caractéristiques de ses paramètres**

- leur nombre
- le type respectif de chacun d'eux

Le compilateur choisira la fonction à utiliser selon les paramètres effectifs par rapport aux paramètres formels des fonctions candidates.

Exemple:

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

// définition de la fonction somme qui calcul la somme de deux réels
double somme(double a, double b)
{  double r;
   r = a + b;
  return r;
}
// définition de la fonction somme qui calcul la somme de deux entiers
double somme(int a, int b)
{  double r;
   r = a + b;
   return r;
}
// définition de la fonction somme qui calcul la somme de trois réels
double somme(double a, double b, double c)
{  double r;
   r = a + b + c;
   return r;
}
```

```
int main()
{  double x, y,z, resultat;
   cout << "Tapez la valeur de x : "; cin >> x;
   cout << "Tapez la valeur de y : "; cin >> y;
   cout << "Tapez la valeur de z : "; cin >> z;

//appel de notre fonction somme (double,double)
  resultat = somme(x, y);
  cout <<  x << " + " << y << " = " << resultat << endl;

//appel de notre fonction somme (int,int)
  resultat = somme(static_cast<int>(x), static_cast<int> ( y));
  cout <<   static_cast<int>(x) << " + " << static_cast<int>( y) << " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(x, y,z);
 cout<<  x << " + " << y << " + " << z<< " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(static_cast<int>(x), static_cast<int>( y), static_cast<int>(z));
 cout<< static_cast<int> ( x) << " + " << static_cast<int>(y) << " + " << static_cast<int>( z)<< " = " <<
resultat << endl;

 return 0;
}
```

Affichage de l'exécution:

```
Tapez la valeur de x : 5.6
Tapez la valeur de y : 7.5
Tapez la valeur de z : 2.3
5.6 + 7.5 = 13.1
5 + 7 = 12
5.6 + 7.5 + 2.3 = 15.4
5 + 7 + 2 = 14
```

III.3 Arguments par défaut

On peut, lors de la déclaration d'une fonction, donner des valeurs par défaut à certains paramètres des fonctions. Ainsi, lorsqu'on appelle une fonction, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres! [4b]

Exemple :

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

 void afficheLigne(const char c, const int n=5)
{  for(int i(0) ; i<n ;  ++i)
    cout<<c ;
 }

int main()
{  afficheLigne('+') ;
   cout<<endl ;
   afficheLigne('*',8) ;
 return 0 ;
}
```

Affichage de l'exécution:

```
+++++
********
Process returned 0 (0x0)   execution time : 3.354 s
Press any key to continue.
```

✍ Il y a quelques règles que vous devez retenir pour les valeurs par défaut:

- Seul le prototype doit contenir les valeurs par défaut.
- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres.
- Vous pouvez rendre tous les paramètres de votre fonction facultatifs.

III.4 Passage des paramètres par valeur et par variable

Il y a deux méthodes pour passer des variables en paramètres dans une fonction: le **passage** par **valeur** et le **passage** par **variable**. Ces méthodes sont décrites ci-dessous.

III.4.1 Passage par valeur

La valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

Pour mieux comprendre, Prenons le programme suivant qui ajoute 2 à l'argument fourni en paramètre.

Exemple :

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

void sp (int );

int main()
{int n = 3;
cout << "n=" << n << endl;
sp (n);
cout << "n=" << n;
return 0 ;
}
void sp (int  nbre)
{   cout << "----------"<<endl ;
   cout << "nbre=" << nbre << endl;
    nbre = nbre + 2;
   cout << "nbre=" << nbre;
   cout << "----------"<<endl ;
}
```

Affichage de l'exécution:

```
n=3
----------
nbre=3
nbre=5
----------
n=3

Process returned 0 (0x0)   execution time : 1.782 s
Press any key to continue.
```

III.4.2 Passage par variable

Consiste à passer l'adresse d'une variable en paramètre. Toute modification du paramètre dans la fonction affecte directement la variable passée en argument correspondant, puisque la fonction accède à l'emplacement mémoire de son argument. Il existe 2 possibilités pour transmettre des **paramètres par variables** :

- Les **références**
- Les **pointeurs**

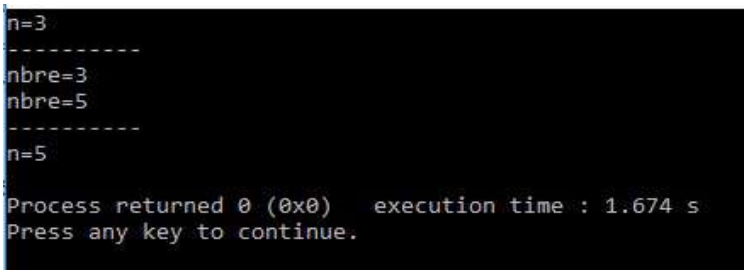
III.4.2.1 Passage par références

Consiste à passer une référence de la variable en argument. Ainsi, aucune variable temporaire ne sera créée par la fonction et toutes les opérations (de la fonction) seront effectuées directement sur la variable. Le plus simple est d'utiliser le mécanisme de référence « **&** ».

Exemple :

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

void sp (int  &);  // ajout de la référence au niveau du prototype

int main()
{int n = 3;
cout << "n=" << n << endl;
sp (n); //appel de la fonction
cout << "n=" << n<<endl ;;
return 0 ;
}
void sp (int &  nbre)       //Ajout de la référence dans la fonction
{   cout << "----------"<<endl ;
    cout << "nbre=" << nbre << endl;
     nbre = nbre + 2;
    cout << "nbre=" << nbre<<endl ;
    cout << "----------"<<endl ;
}
```

Affichage de l'exécution:



III.4.2.2 Passage par adresses (pointeurs)

Consiste à passer l'adresse d'une variable en paramètre. Toute modification du paramètre dans la fonction affecte directement la variable passée en argument correspondant, puisque la fonction accède à l'emplacement mémoire de son argument. Le pointeur indique au compilateur que ce n'est **pas la valeur** qui est transmise, **mais une adresse** (un pointeur).

Exemple :

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

void sp (int  *);  // ajout de l'étoile au niveau du prototype

int main()
{int n = 3;
cout << "n=" << n << endl;
sp (&n); //appel de la fonction
cout << "n=" << n<<endl ;
return 0 ;
}
void sp (int *  nbre)       //Ajout de la référence dans la fonction
{   cout << "----------"<<endl ;
   cout << "nbre=" << *nbre << endl;
    *nbre = *nbre + 2;
   cout << "nbre=" <<* nbre<<endl ;
   cout << "----------"<<endl ;
}
```

Affichage de l'exécution:

```
n=3
----------
nbre=3
nbre=5
----------
n=5

Process returned 0 (0x0)   execution time : 1.674 s
Press any key to continue.
```

III.5 Fonctions *inline* [4b]

Parfois le temps d'exécution d'une fonction est petit comparé au temps nécessaire pour appeler la fonction. Le mot clé ***inline*** informe le compilateur qu'un appel à cette fonction peut être remplacé par le corps de la fonction.

Syntaxe : **inline** type fonct(liste_des_arguments){...}

Exemple

```
inline int max(int a, int b)
{
  return (a < b) ? b : a;
}
void main()
{  int m = max(134, 876);
  cout<< "m=" << m<<endl ;
}
```

- Les fonctions "**inline**" permettent de gagner au niveau de temps d'exécution, mais augmente la taille des programmes en mémoire.
- Contrairement aux macros dans C, les fonctions "**inline**" évitent les effets de bord (dans une macro.
- Les fonctions "**inline**" doivent être définies dans des fichiers d'entête (.h) qui seront inclus dans des fichiers source (.cpp), pour que le compilateur puisse faire l'expansion.

Chapitre IV : Tableaux, Pointeurs et Chaines de caractères en C++

IV.1 Tableaux unidimensionnels

Une variable entière de type ***int*** ne peut contenir qu'une seule valeur. Si on veut stocker en mémoire un ensemble de valeurs, il faut utiliser une structure de données appelée tableau qui consiste à réserver espace de mémoire contiguë dans lequel les éléments du tableau peuvent être rangés. On peut distinguer deux sortes de tableaux unidimensionnels :

- Les tableaux statiques : Ceux dont la taille est connue à l'avance, et
- Les tableaux dynamiques : ceux dont la taille peut varier en permanence.

IV.1.1 Tableaux unidimensionnels statiques

IV.1.1.1 Déclaration et initialisation

Comme toujours en C++, une variable est composée d'un nom et d'un type. Comme les tableaux sont des variables, cette règle reste valable. Il faut juste ajouter une propriété supplémentaire, la taille du tableau.

Syntaxe :

```
type nomDuTableau [taille];
```

- ***type***: définit le type des éléments que contient le tableau.
- ***nomDuTableau***: le nom que l'on décide de donner au tableau.
- ***taille***: nombre entier qui détermine le nombre de cases que le tableau doit comporter.

- Rappelez-vous qu'un tableau en langage C++ est composé uniquement d'éléments de même type.
- Le nom du tableau suit les mêmes règles qu'un nom de variable.

Exemple: [4c]

```
int meilleurScore [5];
char voyelles [6];
double notes [20];
```

Il est possible aussi d'initialiser le tableau à la déclaration en plaçant entre accolades les valeurs, séparées par des virgules.

Exemple :

```
int meilleurScore [5] = {100, 432, 873, 645, 314};
```

- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau.
- Les valeurs entre accolades doivent être des constantes, l'utilisation de variables provoquera une erreur du compilateur.
- Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments sont initialisés à 0.
- Si le nombre de valeur entre accolades est nul, alors tous les éléments du tableau s'initialisent à zéro.

Exemple :

```
int meilleurScore [5] = {};
```

IV.1.1.2 Manipulation des éléments

Un élément du tableau (repéré par le nom du tableau et son indice) peut être manipulé exactement comme une variable, on peut donc effectuer des opérations avec (ou sur) des éléments de tableau.

Le point fort des tableaux, c'est qu'on peut les parcourir en utilisant une boucle. On peut ainsi effectuer une action sur chacune des cases d'un tableau, l'une après l'autre : par exemple afficher le contenu des cases.

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

int main()
{
  int const taille(10); //taille du tableau
 int tableau[taille]; //declaration du tableau

 for (int i(0); i<taille; i++ )
 {  tableau[i]=i*i;     cout<<"Le tableau ["<<i<<"] contient la valeur "<<tableau[i]<<endl;  }
 return 0;
}
```

Affichage de l'exécution:

```
Le tableau [0] contient la valeur 0
Le tableau [1] contient la valeur 1
Le tableau [2] contient la valeur 4
Le tableau [3] contient la valeur 9
Le tableau [4] contient la valeur 16
Le tableau [5] contient la valeur 25
Le tableau [6] contient la valeur 36
Le tableau [7] contient la valeur 49
Le tableau [8] contient la valeur 64
Le tableau [9] contient la valeur 81

Process returned 0 (0x0)   execution time : 1.936 s
Press any key to continue.
```

IV.1.1.3 Tableaux et fonctions

Au même titre que les autres types de variables, on peut passer un tableau comme argument d'une fonction. Voici donc une fonction qui reçoit un tableau en argument :

```
void afficheTableau(int tableau[], int n)
{
 for(int i = 0; i < n; i++)
  cout << tableau[i] << endl;
}
```

À l'intérieur de la fonction *afficheTableau()*, il n'y a aucun moyen de connaître la taille du tableau ! C'est pour cela nous avons ajouté un deuxième argument ***n*** contenant la taille du tableau.

IV.1. 2 Tableaux unidimensionnels dynamiques (classe vector)

Les tableaux que nous avons vus jusqu'ici sont des tableaux statiques. Cette forme de tableaux vient du langage C, et est encore très utilisée. Cependant, elle n'est pas très pratique. En particulier :

- Un tableau de cette forme ne connaît pas sa taille.
- On ne peut pas faire d'affectation globale.
- une fonction ne peut pas retourner de tableaux.

Pour remédier ces trois problèmes, Le C++ a introduit la notion des ***tableaux dynamiques*** ces derniers sont des tableaux dont le nombre de cases peut varier au cours de l'exécution du programme. Ils permettent d'ajuster la taille du tableau au besoin du programmeur.

IV.1. 2.1 Déclaration

La première différence se situe au début de votre programme. Il faut ajouter la ligne **#include** <**vector**> pour utiliser ces tableaux. Et la deuxième différence se situe dans la manière de déclarer un tableau.

Syntaxe:

```
vector <type> nomDuTableau(taille);
```

Par exemple, pour un tableau de 3 entiers, on écrit :

```
vector<int> tab(3);
```

Pour initialiser les éléments de tab2 aux mêmes valeurs que tab1.

```
vector<int> tab2(tab1);
```

On peut même déclarer un tableau sans cases en ne mettant pas de parenthèses du tout:

```
vector<int> tab;
```

IV.1.2.2 Accès aux éléments

On peut accéder aux éléments de tab de la même façon qu'on accéderait aux éléments d'un tableau statique : **tab[0] = 7**.

Exemple: [4c]

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

int main()
{
  int const TailleTab(5);
//Déclaration du tableau
vector<int> Scores(TailleTab);

//Remplissage du tableau
Scores[0] = 100432; Scores[1] = 87347; Scores [2] = 64523; Scores[3] = 31415; Scores[4] = 118218;

for(int i(0); i<TailleTab; ++i)
   cout << "Meilleur score de joueur " << i+1 << " est : " << Scores[i] << endl;
return 0;

}
```

IV.1.2.2 Modification de la taille

On peut modifier la taille d'un tableau soit en ajoutant des cases à la fin d'un tableau ou en supprimant la dernière case d'un tableau.

Commençons par ajouter des cases à la fin d'un tableau.

a. Fonction push_back()

Il faut utiliser la fonction *push_back()*. On écrit le nom du tableau, suivi d'un point et du mot push_back avec, entre parenthèses, la valeur qui va remplir la nouvelle case.

Exemple :

```
vector<int> tableau(3,2);  //Un tableau de 3 entiers valant tous 2
tableau.push_back(8);  //On ajoute une 4ème case au tableau qui contient la valeur 8
```

b. Fonction pop_back()

On peut supprimer la dernière case d'un tableau en utilisant la fonction ***pop_back()*** de la même manière que push_back(), sauf qu'il n'y a rien à mettre entre les parenthèses.

Exemple :

```
vector<int> tableau(3,2);  //Un tableau de 3 entiers valant tous 2
tableau.pop_back(8);  // Il y a plus que 2 éléments dans le tableau
```

IV.1.2.3 Autres opérations de la classe "vector"

Opération	Description
Accès aux éléments, itérateurs	
front()	retourne une référence sur le premier élément **ex** : vf.**front**() += 11; // +11 au premier élément
back()	retourne une référence sur le dernier élément **ex :** vf.**back**()+=22; // +22 au dernier élément
at()	méthode d'accès avec contrôle de l'existence de l'élément (possibilité de récupérer une erreur en cas de débordement) **ex : for**(**int** i=0; i<vi.**size**(); i++) vi.**at**(i)=rand()%100;
data()	retourne un pointeur sur le premier élément du tableau interne au conteneur **ex: int***p2=vi.**data**(); **for**(**int** i=0; i<vi.**size**(); i++, p2++) **cout**<<*p2<<"-"; **cout**<<endl;
Affectation, Insertion et Suppression d'éléments n'importe où dans le tableau	

assign()	remplace le contenu d'un vecteur par un nouveau contenu en adaptant sa taille si besoin
	ex:
	vector<int> v1; **vector**<int> v2;
	v1.**assign**(10, 50); // v1 remplacé par 10 entiers à 50
	int tab[] = { 10, 20, 30 }; // v2 remplacé par les éléments du tableau tab
	v2.**assign**(tab, tab + 3);
	// affichage des tailles des vecteurs
	cout << **int**(v1.**size**()) << **endl**; cout << **int**(v2.**size**()) << **endl;**

insert(p, x)	ajoute un élément **x** avant l'élément désigné par l'itérateur **p** **ex:** **vector**<**float**> v; v.**insert**(v.**begin**(),1.5); // ajoute 1.5 avant, au début
insert(p, n, x)	ajoute n copies de **x** avant l'élément désigné par l'itérateur **p** **ex:** v.**insert**(v.**end**(),5,20) ; // ajoute cinq éléments initialisés 20 à la fin

emplace(p,x)	ajoute un élément x à la position désignée par l'itérateur p et décale le reste. Les arguments passés pour x correspondent à des arguments pour le constructeur de x
	ex : p=v.**begin**()+2; // ajoute 100 à la position 2
	v.**emplace**(p, 100);

erase(p)	supprime l'élément pointé par l'itérateur **p** **ex:** //supprime les éléments compris entre les itérateurs premier et dernier **vector**<**float**>::iterator prem = v.**begin**()+1; **vector**<**float**>::iterator dern = v.**end()-**1; v.**erase**(prem,dern); affiche_vector(v);
clear()	efface tous les éléments d'un conteneur. Équivalent à c.**erase**(c.begin(), c.end()) **ex:** v.**clear**(); // efface tout le conteneur
Taille et capacité	
size()	retourne le nombre d'éléments du « vector » **ex :** **vector**<**int**> v(5); **cout**<<v.**size**()<<endl; // 5

resize(nb)
resize(nb, val)
	redimensionne un « vector » avec nb éléments. Si le conteneur existe avec une taille plus petite, les éléments conservés restent inchangés et les éléments supprimés sont perdus. Avec une taille plus grande, les éléments ajoutés sont initialisés avec une valeur par défaut ou avec une valeur spécifiée en val
	ex: **vector**<**int**> **v**(5);
	for(unsigned i=0; i<v.**size**(); i++)
	v[i]=i;
	affiche_vector(v); // 0,1,2,3,4
	v.**resize**(7);
	affiche_vector(v); // 0,1,2,3,4,0,0
	v.**resize**(10,99);
	affiche_vector(v); // 0,1,2,3,4,0,0,99,99,99
	v.**resize**(3);
	affiche_vector(v); // 0,1,2

capacity()	retourne le nombre courant d'emplacements mémoire réservés. C'est-à-dire le total de mémoire allouée en nombre d'éléments pour le conteneur. Attention, à ne pas confondre avec le nombre des éléments effectivement contenus retourné par size().
	ex : **vector**<**int**>v(10);
	cout<<"nombre elements : "<<v.**size**()<<endl;//10
	cout<<"capacite : "<<v.**capacity**()<<endl; // 10
	v.**resize**(12);
	cout<<"nombre elements : "<<v.**size**()<<endl; //12
	cout<<"capacite : "<<v.**capacity**()<<endl; // 15

Le parcours d'un « vector » peut aussi s'effectuer en utilisant des itérateurs (pointeurs) plutôt que le système d'indice. De ce fait, la classe « vector » est équipée avec toutes les méthodes les concernant :

Méthode	Description
begin()	retourne un itérateur « iterator » qui pointe sur le premier élément
end()	retourne un itérateur « iterator » qui pointe sur l'élément suivant le dernier
rbegin()	retourne un itérateur « reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier) ;
rend()	retourne un itérateur « reverse_iterator » qui pointe sur l'élément suivant le dernier dans l'ordre inverse (balise de fin, par exemple NULL)
cbegin()	retourne un itérateur « const_iterator » qui pointe sur le premier élément. L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié
cend()	retourne un itérateur « const_iterator » qui pointe sur ce qui suit le dernier élément (balise de fin, par exemple NULL). L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié
crbegin()	retourne un itérateur « const_reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié
crend()	retourne un itérateur « const_reverse_iterator » qui pointe sur la fin de la séquence inverse, avant le premier élément (balise de fin sens inverse, par exemple NULL). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié

Exemple :

```
vector<int>vi(15, 7);  // 15 cases entières initialisées avec 7
vector<int>::iterator it;
   for (it=vi.begin(); it!=vi.end(); it++)
      cout<<*it<<"-";
   cout<<endl;
```

IV.1.2.3 Les vector et les fonctions

Passer un tableau dynamique en argument à une fonction est beaucoup plus simple que pour les tableaux statiques. Comme pour n'importe quel autre type, il suffit de mettre **vector<type>** en argument. Et c'est tout. Grâce à la fonction **size()**, il n'y a même pas besoin d'ajouter un deuxième argument pour la taille du tableau :

Exemple :

```
void afficheTableau(vector<int> tab)
{  for(int i = 0; i < tab.size(); i++)
     cout << tab[i] << endl;
}
```

✍ Si le tableau contient beaucoup d'éléments, le copier prendra du temps. Il vaut donc mieux utiliser un passage par référence constante const& pour optimiser la copie. Ce qui donne:

```
void afficheTableau(vector<int> const& tab)
{
      // ...
}
```

Dans ce cas, le tableau dynamique ne peut pas être modifié. Pour changer le contenu du tableau, il faut utiliser un passage par référence tout simple (sans le mot const donc). Ce qui donne :

```
void afficheTableau(vector<int>& tab)
{ … }
```

Pour appeler une fonction recevant un **vector** en argument, il suffit de mettre le nom du tableau dynamique comme paramètre entre les parenthèses lors de l'appel. Ce qui donne :

```
vector<int> tab(3,2);       //On crée un tableau de 3 entiers valant 2
afficheTableau(tab);       //On passe le tableau à la fonction afficherTableau()
```

Il est possible d'écrire une fonction renvoyant un vector.

```
vector<int> premierCarres(int n)
{   vector<int> tab(n);
   for(int i = 0; i < tab.size(); i++)
     tab[i] = i * i;
 return tab;
}
```

IV.2 Tableaux multidimensionnels dynamiques

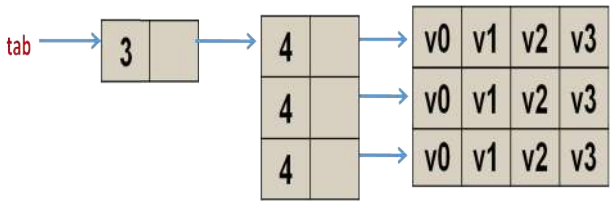
Notez qu'il est aussi possible de créer des tableaux multi-dimensionnels de taille variable en utilisant les **vector**s.

Pour un tableau 2D d'entiers, on devra écrire:

```
vector < vector<int> > tab;  // un vecteur de vecteurs :« tab » qui ne contient aucun élément
```

Pour dimensionner ce vecteur de vecteurs, il faudra le faire en 2 fois:

```
tab.resize(3); // 3 vecteurs de vecteurs vide pour l'instant
for (int ligne=0; ligne<tab.size(); ligne++ )
   tab [ligne].resize (4); // dimensionner chacun des vecteurs imbriqués
```



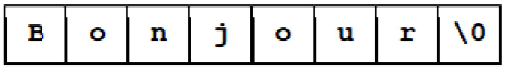
Cela nous permet de créer un vecteur de 3 éléments désignant chacun 1 vecteur de 4 éléments. Finalement, on peut accéder aux valeurs dans les cases de du tableau en utilisant deux paires de crochets tab[i][j], comme pour les tableaux statiques. Il faut par contre s'assurer que cette ligne et cette colonne existent réellement.

Exemple :

```
vector<vector<int> > matrice;
matrice.push_back(vector<int>(5));   //On ajoute une ligne de 5 cases à notre matrice
matrice.push_back(vector<int>(3,4)); //On ajoute une ligne de 3 cases contenant chacune 4
matrice[0].push_back(8);   //Ajoute une case contenant 8 à la première ligne de la matrice
```

IV.3 Les strings sont des tableaux de caractères (lettres)

En C++, une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, avec un caractère nul '\0' marquant la fin de la chaîne.
On peut par exemple représenter la chaîne « Bonjour » de la manière suivante :



Les chaînes de caractère peuvent être saisies au clavier et affichées à l'écran grâce aux objets habituels **cin** et **cout** .

IV.3.1 Déclaration

Pour définir une chaîne de caractères en langage C, il suffit de définir un tableau de caractères. Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne).

Exemple :

```
char nom[20], prenom[20];    // 19 caractères utiles
char adresse[3][40];     // trois lignes de 39 caractères utiles
char texte[10] = {'B','o', 'n', 'j','o', 'u','r', '\0'};     // ou : char texte[10] = "Bonjour";
```

✍ On peut utiliser un **vector** si l'on souhaite changer la longueur du texte :

```
vector<char> texte;
```

✍ En théorie, on pourrait donc se débrouiller en utilisant des tableaux statiques ou dynamiques de **char** à chaque fois que l'on veut manipuler du texte. Mais ce serait fastidieux. C'est pour ça que les concepteurs du langage ont décidé de cacher tout ces mécanismes dans une boîte fermée (Objet) en utilisant la classe **string**. Cette dernière, propose en fait une encapsulation de la chaîne de caractères C. La bibliothèque standard du C++ propose d'autres types de chaînes de caractères, notamment celles qui sont capables de gérer un encodage comme l'UTF-8 où un caractère est stocké sur plusieurs octets.

IV.3.2 Création d'objets « string »

La création d'un objet ressemble beaucoup à la création d'une variable classique comme **int** ou **double**:

```
#include <iostream>
#include <string>     // Obligatoire pour pouvoir utiliser les objets string
using namespace std;
int main()
{
   string maChaine;  //Création d'un objet 'maChaine' de type string
   return 0;
}
```

IV.3.3 Instanciation et initialisation de chaînes

Pour initialiser notre objet au moment de la déclaration (et donc lui donner une valeur !), il y a plusieurs possibilités:

```
int main()
{   string maChaine("Bonjour !");    //Création d'un objet 'maChaine' de type string et initialisation
    String  s3 = "chaine 3";
    return 0;
}
```

IV.3.4 Accès à un caractère

On manipule une chaîne C++ comme une chaîne C : on peut utiliser les crochets pour accéder à un élément même si la chaîne C++ n'est pas un tableau. Nous verrons plus tard comme on peut faire cela.

```
// lecture d'un caractère
cout << s2[3];        // 4eme caractère
cout << s2.at(3);   // 4eme caractère
// modification de caractère
s2[2]    = 'A';
s2.at(3) = 'B';
```

IV.3.5 Concaténation de chaînes

Cette opération permet d'assembler deux chaines de caractères:

```
#include <iostream>
#include<string>
 using namespace std;
 int main (void)
{   string s1, s2, s3;
     cout << "Tapez une chaine : "; getline (cin, s1);
     cout << "Tapez une chaine : "; getline (cin, s2);
     s3 = s1 + s2;
     cout << "Voici la concatenation des 2 chaines :" << endl;
     cout << s3 << endl;
  return 0;
}
```

✍ Par défaut, lorsqu'on saisit une chaîne de caractères en utilisant cin, le séparateur est l'espace : cela empêche de saisir une chaîne de caractères comportant un espace.

✍ La fonction **getline(iostream &,string)** permet de saisir une chaîne de caractères en utilisant le passage à la ligne comme séparateur : notre chaîne de caractères peut alors comporter des espaces.

✍ On peut utiliser une autre syntaxe de concaténation: s1.**append** (s2) qui est équivalente à s1 = s1+s2

IV.3.6 Quelques méthodes utiles du type « string » [16]

IV.3.6.1 Méthode size()

La méthode **size()** permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type **string**.

Exemple :

```
int main()
{   string maChaine("Bonjour !");
   cout << "Longueur de la chaine : " << maChaine.size();
    return 0;
}
```

```
Longueur de la chaine : 9
```

Affichage de l'exécution

IV.3.6.2 Méthode « erase() »

Cette méthode très simple supprime tout le contenu de la chaîne :

Exemple :

```
int main()
{    string chaine("Bonjour !");
    chaine.erase(0, 4);   // efface les 4 premiers caractères
    cout << "La chaine contient : " << chaine << endl;
    chaine.erase();
    cout << "La chaine contient : " << chaine << endl;
    return 0;
}
```

Affichage de l'exécution:

```
La chaine contient :our !
La chaine contient :
```

IV.3.6.3 Méthode « substr() »

Une autre méthode peut se révéler utile: **substr()**. Elle permet d'extraire une partie de la chaîne stockée dans un string.

Syntaxe : string **substr**(size_type index, size_type num = npos);

- **Index**: permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère).
- **Num** : permet d'indiquer le nombre de caractères que l'on prend. Par défaut, la valeur est ***npos***, ce qui revient à prendre tous les caractères qui restent. Si vous indiquez 2, la méthode ne renverra que 2 caractères.

Exemple:

```
int main()
{    string chaine("Bonjour !");
    cout << chaine.substr(3) << endl;
    return 0;
}
```

```
Jour !
```

Affichage de l'exécution

```
int main()
{   string chaine("Bonjour !");
    cout << chaine.substr(3, 4) << endl;
    return 0;
}
```

```
Jour
```

Affichage de l'exécution

✍ On a demandé à prendre 4 caractères en partant du caractère n°3, ce qui fait qu'on a récupéré « jour ».

IV.3.6.4 Méthode «c_str() »

Cette méthode permet de renvoyer un pointeur vers le tableau de char que contient l'objet de type string.

- **Transformation de chaîne de type C en string:** on peut utiliser le constructeur string(char *) ou l'affectation grâce au symbole = d'un char * vers une string.
- **Transformation d'un string en chaîne de type C :** il suffit d'utiliser la méthode : **c_str()** qui renvoie un char * qui est une chaîne de type C.

Exemple:

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

int main (void)
{   string s1, s2;
    char c1 []= "BONJOUR";
    const char * c2;
    s1 = c1;
    cout << s1 << endl;
    s2 = "AU REVOIR";
    c2 = s2.c_str();
    cout << c2 << endl;
  return 0;
 }
```

```
BONJOUR
AU REVOIR
```

Affichage de l'exécution

✍ Dans cet exemple, c1 est un tableau de 8 char contenant la chaîne "BONJOUR " sans oublier le caractère de fin de chaîne '\0'.

✍ Le pointeur c2 est un pointeur vers un tableau non modifiable de char.

✍ Les variables s1 et s2 sont des string. On peut affecter directement s1=c1 : le tableau de char sera transformé en string.

✍ Dans c2, on peut récupérer une chaîne « de type C » identique à notre string en écrivant c2=s2.**c_str()**. On peut transformer aisément un string en tableau de char et inversement.

IV.3.6.5 Méthode «istr() »

- Pour transformer une chaîne en double ou en int, il faut transformer la chaîne en flot de sortie caractères : il s'agit d'un **istringstream**. Ensuite, nous pourrons lire ce flot de caractères en utilisant les opérateurs usuels >>.
- La méthode **eof()** sur un **istringstream** permet de savoir si la fin de la chaîne a été atteinte. Chaque lecture sur ce flot grâce à l'opérateur >> renvoie un booléen qui nous indique d'éventuelles erreurs.

Exemple :

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main (void)
 {
  string s;
  cout << "Tapez une chaine : "; getline (cin, s);
  istringstream istr(s);
  int i;

  if (istr >> i) cout << "VOUS AVEZ TAPE L'ENTIER " << i << endl;
  else cout << "VALEUR INCORRECTE" << endl;
  return 0;
 }
```

```
Tapez une chaîne : 12345
VOUS AVEZ TAPE L'ENTIER 12345
```

Affichage de l'exécution

✍ Dans cet exemple, ***s*** est une chaîne : on saisit une chaîne au clavier en utilisant getline(cin,s). On crée ensuite un **istringstream** appelé ***istr*** et construit à partir de ***s***.

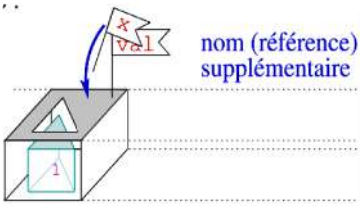
✍ On peut lire un entier ***i*** à partir de ***istr*** en utilisant : istr>>i . (istr>>i) renvoie true si un entier valide a pu être lu et renvoie false sinon. De la même manière, on pourrait lire des données d'autres types, double par exemple.

IV.3.6.6 Autres opérations : insert, replace, find

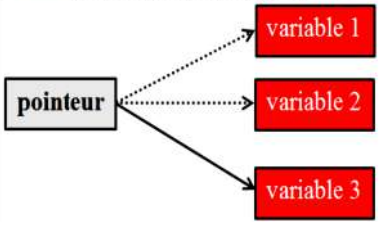
```
s1.insert(3, s2);   // insère s2 à la position 3
s3.replace(2,3,s1);   // remplace la portion de s3 définie de la position 2 à 5 par la chaîne s1
unsigned int i = s.find("trouve", 4);  // recherche "trouve" à partir de la 4ème position, renvoie
                                       //   std::string::npos le cas échéant
```

IV.4 Pointeurs et allocation dynamique [9]

Dans un programme, pour garder un lien vers une donnée (une variable), on utilise des références ou des pointeurs. En programmation, les pointeurs et références servent essentiellement à trois choses:



1. Permettre à plusieurs portions de code de partager des objets (données, fonctions,..) sans les dupliquer
   => **Référence**

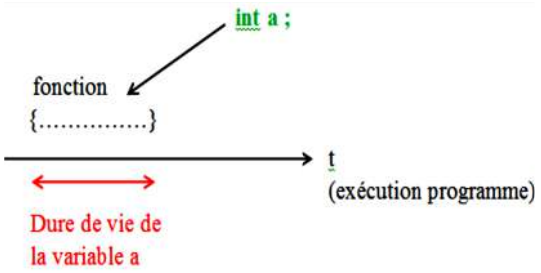


2. Pouvoir choisir des éléments non connus a priori (au moment de la programmation)
   => **Généricité**

3. Pouvoir manipuler des objets dont la durée de vie dépasse la portée

   => **Allocation dynamique**

   La durée de vie de la variable a est égale au temps d'exécution de sa portée.



IV.4.1 Différents pointeurs et références [9]

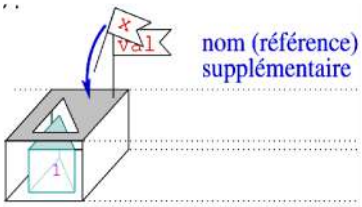
En C++, il existe plusieurs sortes de « pointeurs » (pointeur et références) :

- **Les références:** totalement gérées en interne par le compilateur. Très sûres, donc; mais sont fondamentalement différentes des vrais pointeurs.
- **Les « pointeurs intelligents » (smart pointers):** gérés par le programmeur, mais avec des gardes-fous. Il en existe 3: *unique_ptr*, *shared_ptr, weak_ptr* (avec #include <memory>)
- **Les « pointeurs « hérités de C » » (build-in pointers):** les plus puissants (peuvent tout faire) mais les plus « dangereux »

IV.4.1.1 Référence

Une référence est un autre nom pour un objet existant, un synonyme, un alias.
La syntaxe de déclaration d'une référence est: **<type>& nom_reference(identificateur) ;**
Après une telle déclaration, **nom_reference** peut être utilisé partout où **identificateur** peut l'être [9].

```
int val(1) ;
int& x(val) ;
```



✍ C'est exactement ce que l'on utilise lors d'un passage par référence:

Exemple 1 :

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

void f(int & a )
{ ++a ; }

int main (void)
 {  int b=1;
    f(b) ;
    cout << " APRES b =  "<<b;
  return 0;
 }
```

```
 APRES b =  2
Process returned 0 (0x0)   execution time : 2.728 s
Press any key to continue.
```

Résultat de l'exécution

Exemple 2 : **Sémantique de const**

```
int i(3);
const int &   j(i);     // i et j sont les mêmes. On ne peut pas changer la valeur via j
// j=12 ;       // Erreur de compilation
i = 12 ;       // oui, et j aussi vaut 12
cout << " j =  "<<j;
```

```
j = 12
Process returned 0 (0x0)
Press any key to continue.
```

Résultat de l'exécution

✍ **Spécificités des références :** Contrairement aux pointeurs, une référence :

a. doit absolument être initialisée (vers un objet existant) :

```
int i;
int &  rj;     // Non, la référence rj doit être liée à un objet !
```

b. ne peut être liée qu'à un seul objet :

```
int i;
int &  ri;
int j(2) ;
ri = j ;    /* Ne veut pas dire que ri est maintenant un alias de j mais que
                                                    i prend la valeur de j */
j = 3 ;
cout<< i <<endl ; // affiche 2
```

c. ne peut pas être référencée

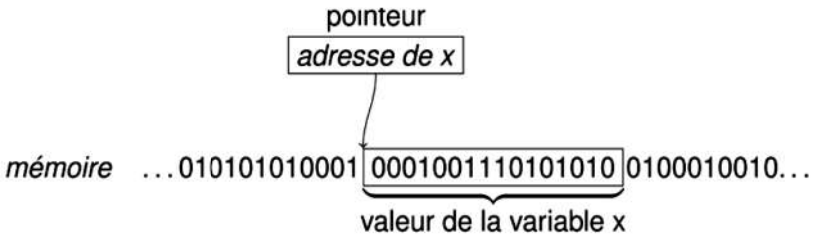
```
int i(3);
int &  ri(i);
int &  rri(ri); // Non
int &&  rri(ri); // Non plus !
```

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: conflicting declaration 'int&& rri'
error: 'rri' has a previous declaration as 'int& rri'
warning: unused variable 'rri' [-Wunused-variable]
=== Build failed: 2 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===
```

⇒ On ne peut donc pas faire de tableau de références

IV.4.1.2 Pointeurs [9]

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.
Un **pointeur** est une variable qui contient **l'adresse** d'un autre objet informatique (par ex, variable) => une « variable de variable »



Notes :

- Une référence n'est pas un «vrai pointeur» car ce n'est pas une variable en tant que telle.
- Une référence est «une autre étiquette».
- Un pointeur «une variable contenant une adresse»

✍ La déclaration d'un pointeur se fait selon la syntaxe suivante : **type* identificateur ;**
Cette instruction déclare une variable de nom **identificateur** de type pointeur sur une valeur de type **type**.
Exemple: déclare une variable **ptr** qui pointe sur une valeur de type int: **int*** ptr ;

✍ L'initialisation d'un pointeur se fait selon la syntaxe suivante:

type* identificateur(adresse) ;

Exemple:

```
Int* ptr(NULL);
int *  ptr(&i);
int *  ptr(new int(33));
```

nullptr : mot clé C++, spécifie que le pointeur ne pointe sur rien

- Pour le compilateur, une variable est un emplacement dans la mémoire,
- Cet emplacement est identifié par une adresse
- Chaque variable possède une et une seule adresse.
- Un pointeur permet d'accéder à une variable par son adresse.
- Les pointeurs sont des variables faites pour contenir des adresses.

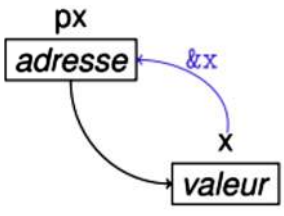
IV.4.1.2.1 Opérateurs sur les pointeurs

C++ possède deux opérateurs particuliers en relation avec les pointeurs : **&** et ***.

- **&** est l'opérateur qui retourne l'adresse mémoire de la valeur d'une variable. Si x est de type **type**, &x est de type **type*** (pointeur sur type).

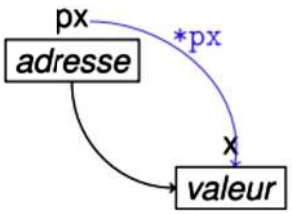
```
int  x(3);
int *  px(NULL);
px = &x;
cout<< "Le pointeur px contient l'adresse"<<px<<endl ;
```

```
Le pointeur px contient l'adresse 0x28fee8
```



- ***** est l'opérateur qui retourne la valeur pointée par une variable pointeur. Si px est de type **type***, (*px) est la valeur de type **type** pointée par px.

```
int  x(3);
int *  px(NULL);
px = &x;
cout<< "Le pointeur px contient l'adresse"<<*px<<endl ;
```




```
Le pointeur px pointe sur la valeur 3
```

Note : ***&i** est donc strictement équivalent à **i**

Attention aux confusions

- **int& id(i)** - id est une référence sur la variable entière i
- **&id** est l'adresse de la variable id

```
int  i(3);
int&  id(i);
cout<< " i = "<< i <<" id = "<< id <<endl ;
cout<< " Adresse de i = "<< &i  <<endl ;
```

```
i = 3 id = 3
adresse de i = 0x28fee8
```

- **int* id**; déclare une variable id comme un pointeur sur un entier
- ***id** (où id est un pointeur) représente le contenu de l'endroit pointé par id

```
int*  ptr(NULL);
int  i = 2;
p = &i ;
cout<< " i = "<< i <<" id = "<< id <<endl ;
cout<< " Le pointeur p pointe sur la valeur  "<< *p <<endl ;
```

```
Le pointeur p pointe sur la valeur 2
```

IV.4.1.2.2 Pointeurs et Tableaux

Par convention, le nom d'une **variable** utilisé dans une partie droite d'expression donne le **contenu** de cette **variable** dans le cas d'une variable simple. Mais un **nom de tableau** donne l'**adresse du tableau** qui est l'adresse du premier élément du tableau.

Nous faisons les constatations suivantes :

- un **tableau** est une ***constante d'adressage*** ;
- un **pointeur** est une ***variable d'adressage***.

Ceci nous amène à regarder l'utilisation de pointeurs pour manipuler des tableaux, en prenant les variables: long i, tab[10], *pti ;

- tab est l'adresse du tableau (adresse du premier élément du tableau &tab[0] ;
- **pti = tab ;** initialise le pointeur pti avec l'adresse du début de tableau. Le & ne sert à rien dans le cas d'un tableau. pti = &tab est inutile et d'ailleurs non reconnu ou ignoré par certains compilateurs ;
- **&tab[1]** est l'adresse du 2e élément du tableau.
- **pti = &tab[1]** est équivalent à: pti = tab ; où pti pointe sur le 1er élément du tableau.

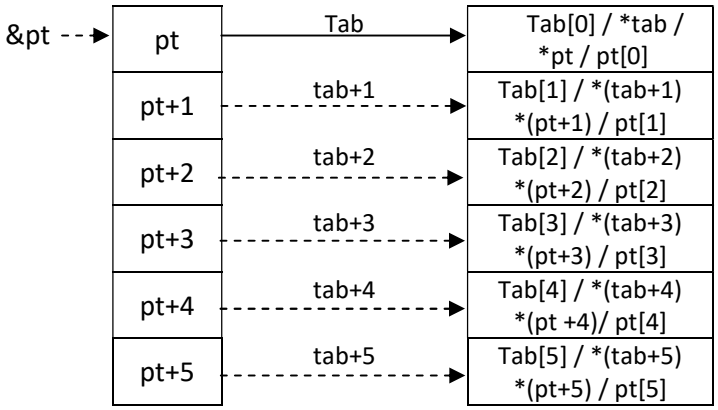
- **pti += 1 ;** fait avancer, le pointeur d'une case ce qui fait qu'il contient l'adresse du 2ème élément du tableau.

Nous pouvons déduire de cette arithmétique de pointeur que :

- **tab[i]** est équivalent à ***(tab +i)**.
- ***(pti+i)** est équivalent à **pti[i]**.

La figure suivante est un exemple dans lequel sont décrites les différentes façons d'accéder aux éléments d'un tableau **tab** et du pointeur **pt** après la définition suivante:

int tab[6], *****pt = tab;

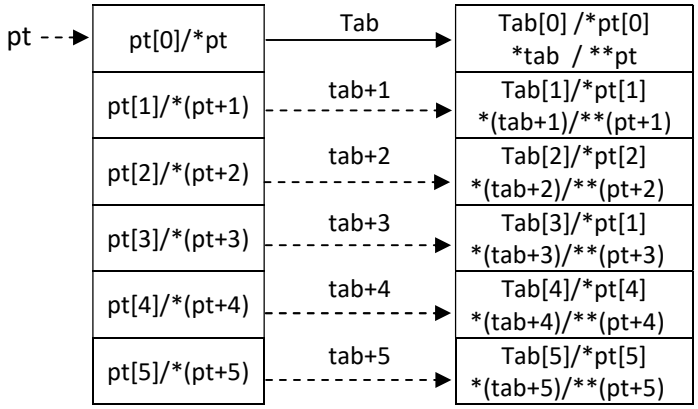


IV.4.1.2.3 Tableau de pointeurs

La définition d'un tableau de pointeurs se fait par : **type ***nom[taille];

Le premier cas d'utilisation d'un tel tableau est celui où les éléments du tableau de pointeurs contiennent les adresses des éléments du tableau de variables. La figure suivante illustre un exemple d'utilisation de tableau de pointeurs à partir des définitions de variables suivantes :

int tab[3], *****pt[6] = {tab,tab+1,tab+2,tab+3,tab+4,tab+5};



IV.4.2 Allocation dynamique

Il y a trois façons d'allouer de la mémoire en C++

1. **Allocation statique :** La réservation mémoire est déterminée à la **compilation.**
   L'espace alloué statiquement est déjà réservé dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécuter. L'avantage de l'allocation statique se situe essentiellement au niveau des performances, puisqu'on évite les coûts de l'allocation dynamique à l'exécution.

2. **Allocation dynamique:** La mémoire est réservée pendant **l'exécution** du programme.
   L'espace alloué dynamiquement ne se trouve pas dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécute.
   Par exemple, les tableaux de taille variable (**vector**), les chaînes de caractères de type **string**.

Dans le cas **particulier des pointeurs**, l'allocation dynamique permet également de réserver de la mémoire **indépendamment de toute variable**: on pointe directement sur une zone mémoire plutôt que sur une variable existante.

IV.4.2.1 Allocation d'une case mémoire

C++ possède deux opérateurs **new** et **delete** permettant **d'allouer** et de **libérer dynamiquement de la mémoire**.

Syntaxe: pointeur = **new** type ⇒ Réserve une zone mémoire de type **type** et affecte l'adresse dans la variable **pointeur.**

Exemple :

```
int*  p;
p = new int ;
*p = 2 ;
cout<< " p = "<< p <<" *p = "<< *p <<endl ;
```

```
p = 0x350cf0 *p = 2
Process returned 0 (0x0)
Press any key to continue.
```

```
int*  p;
p = new int (2);
cout<< " p = "<< p <<" *p = "<< *p <<endl ;
```

IV.4.2.2 Libérer la mémoire allouée

Syntaxe : delete pointeur ⇒ libère la zone mémoire allouée au pointeur

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose. **On ne peut plus y accéder !**

Bonnes pratiques

- Faire suivre tous les **delete** de l'instruction **« pointeur = NULL; »**.
- Toute zone mémoire allouée par un **new** doit impérativement être libérée par un **delete** correspondant !

Exemple :

```
int*  px(NULL);
px = new int ;
*px = 20 ;
cout<< "*px = "<<*px ;
delete px ;
px = NULL ;
```

Notes :

- Une variable allouée statiquement est désallouée automatiquement (à la fermeture du bloc).
- Une variable (zone mémoire) allouée dynamiquement doit être désallouée explicitement par le programmeur.

```
int*  px(NULL);
{px = new int(4) ; int n=6 ;}
cout<< "*px = "<<*px ;
delete px ; px = NULL ;
cout<< "*px = "<<*px <<endl;
cout<< "n = "<< n ;
```

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
warning: unused variable 'n' [-Wunused-variable]
error: 'n' was not declared in this scope
=== Build failed: 1 error(s), 1 warning(s) (0 minute(s), 0 second(s))
```

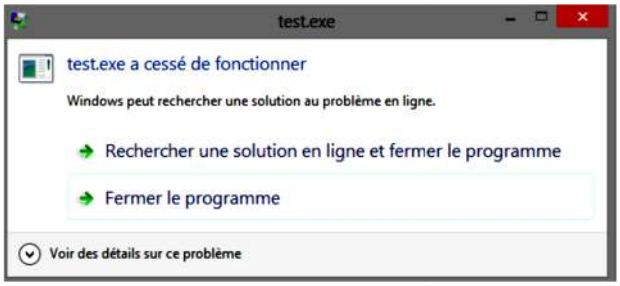
✍ Attention

Si on essaye d'utiliser la valeur pointée par un pointeur pour lequel aucune mémoire n'a été réservée, une erreur de type **Segmentation fault** se produira à l'exécution.

```
int*  px;
*px = 20 ; // ! Erreur : px n'a pas été alloué
cout<< "*px = "<<*px<<endl;
```

Compilation : Ok

Exécution: arrêt programme (Segmentation fault)



✍ Bonnes pratiques

```
int*  px(NULL);
if(px != NULL)
{   *px = 20 ;
   cout<< "*px = "<<*px<<endl;
}
```

Initialisez toujours vos pointeurs

Utilisez **NULL** si vous ne connaissez pas encore la mémoire pointée au moment de l'initialisation

IV.4.2.3 Allocation dynamique de tableau unidimensionnel

Syntaxe*:* **new** nom_type **[**n**]**

⇒ Permet d'allouer dynamiquement un espace mémoire nécessaire pour un tableau de *n* éléments de type *nom_type.*

Syntaxe : **delete []** adresse

⇒Permet de libérer l'emplacement mémoire désigné par *adresse* alloué préalablement par ***new***[]

IV.4.2.4 Allocation dynamique de tableau bidimensionnel

Syntaxe :

```
type **data;
  data = new type*[ligne];       // Construction des lignes
   for (int j = 0; j < ligne; j++)
      data[j] = new  type[colonne];  // Construction des colonnes
```

⇒ Permet d'allouer dynamiquement un espace mémoire nécessaire pour un tableau de ligne × colonne éléments de type ***type***.

Syntaxe :

```
for (int i = 0; i < ligm;  i++)
     delete[lignes] data[i];              // Suppression des colonnes
  delete[] data;                    // Suppression des lignes
```

⇒ Permet de libérer l'emplacement mémoire désigné par data alloué préalablement par ***new***[]

Exemple 1:

```
#include <iostream>
#include <ctime>
Using namespace std;

const int MAX =50;
Const int MIN =25;

void display(double **data)
{  for (int i = 0; i < m; i++)
    { for (int j = 0; j < n; j++)
         cout << data[i][j] << " ";
     cout << "\n" << endl;
    }
 }

void de_allocate(double **data)
{ for (int i = 0; i < m;  i++)
     delete[] data[i];
  delete[] data;
}

int main(void)
{   double **data;
   int m,n;    //m: lignes , n:colonnes
    srand(time(NULL));

    cout<<"Colonnes : n = " ; cin>>n;
    cout<< " Lignes :  m = " ; cin>>m;
```

```
    data = new double*[m];
     for (int j = 0; j < m; j++)
        data[j] = new  double[n];

    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
         data[i][j] = (rand() % (MAX - MIN + 1)) + MIN;

  display(data);
  de_allocate(data);
  return 0;
  }
```

Exemple 2:

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

struct Etudiant
{ string nom ;
  double moyenne ;
  char sexe ;  // 'F'  ou 'M'
} ;
void initialiserEtudiant(Etudiant *e) ;
void afficherEtudiant(Etudiant *e) ;

int main(void)
{   Etudiant *ptrEtudiant ;

    ptrEtudiant = new Etudiant;
  initialiserEtudiant(ptrEtudiant) ;
  afficherEtudiant(ptrEtudiant) ;

  delete (ptrEtudiant) ;
  return 0;
  }
void initialiserEtudiant(Etudiant *e)
{    cout<< " Saisir le nom : " ;  cin>>(*e).nom ;
     cout<< " Saisir la moyenne : " ;  cin>>(*e).moyenne ;
     cout<< " Saisir le sexe : " ;  cin>>(*e).sexe ;
}
void  afficherEtudiant(Etudiant *e)
{   if((*e).sexe== 'F')  cout<< " Madame " ;
   else cout<< " Monsieur " ;
   cout<< (*e).nom <<" Votre moyenne est de  "<< (*e).moyenne<<endl ;
}
```

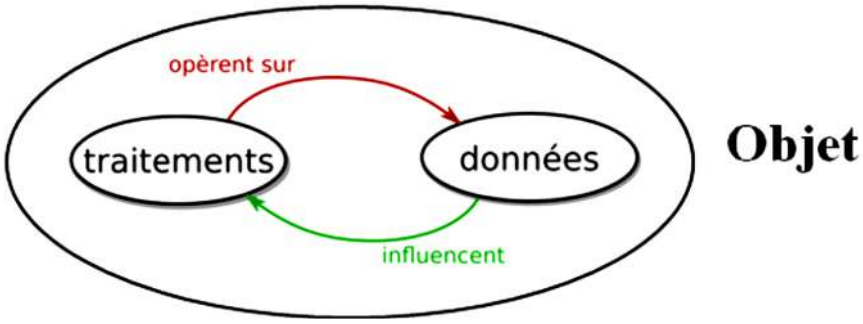
Chapitre V : Classes et objets

V.1 Concept objet

La **POO** permet d'améliorer [9]:

- La robustesse (changement de spécification)
- La modularité
- Lisibilité

} => la maintenabilité



POO : quatre concepts de base
Un des objectifs principaux de la notion d'objet : organiser des programmes complexes grâce aux notions :

- d'encapsulation
- d'abstraction
- d'héritage
- et de polymorphisme

V.2 Notion d'objet (instance):

Un ***objet*** représente une entité individuelle et identifiable, réelle ou abstraite, avec un rôle bien défini dans le domaine du problème, chaque objet peut être caractérisé par une identité, des états significatifs et par un comportement.

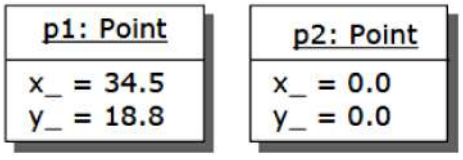
Objet = Etat + Comportement + Identité

V.2.1 Etat

- Un attribut est une caractéristique, qualité ou trait intrinsèque qui contribue à faire unique un objet
- L'état d'un objet comprend les propriétés statiques (attributs) et les valeurs de ces attributs qui peuvent être statiques ou dynamiques

Exemple:

Les attributs de l'objet point en deux dimensions sont les coordonnées x et y

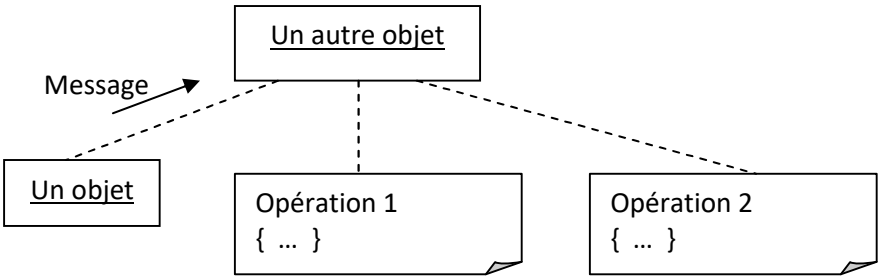


V.2.2 Comportement

Le comportement d'un objet se défini par l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés (un message = demande d'exécution d'une opération) par les autres objets.

Exemple:

Un point peut être déplacé, tourné autour d'un autre point, etc.



V.2.3 Identité

Propriété d'un objet qui permet de le distinguer des autres objets de la même classe.

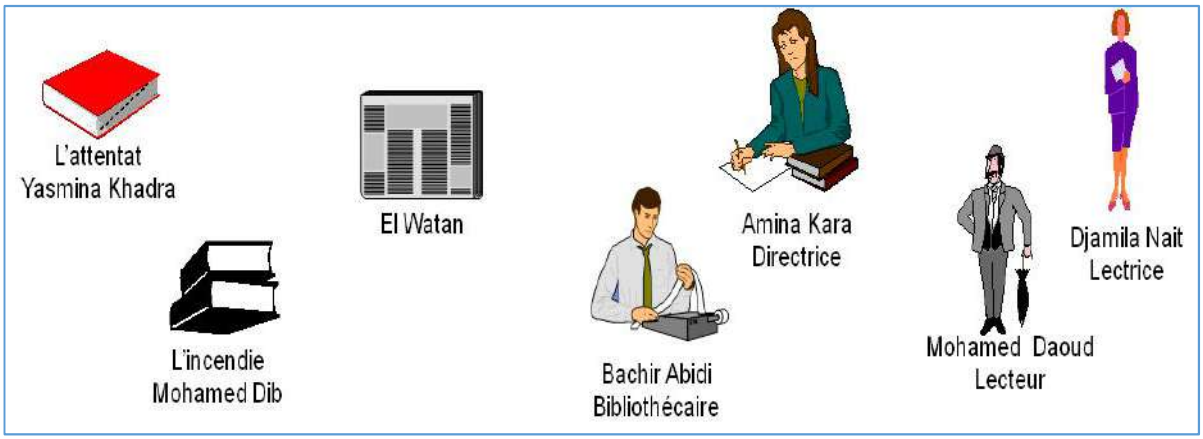
V.3 Notion de classe:

Lorsque des objets ont les mêmes attributs et comportements: ils sont regroupés dans une famille appelée**: Classe.**

Une classe est un modèle à partir duquel on peut générer un ensemble d'objets partageant des attributs et des méthodes communes. Les objets appartenant à celle-ci sont les **instances** de cette classe c.à.d. que **L'instanciation** est la création d'un objet d'une classe.

Exemple: Gestion d'une bibliothèque

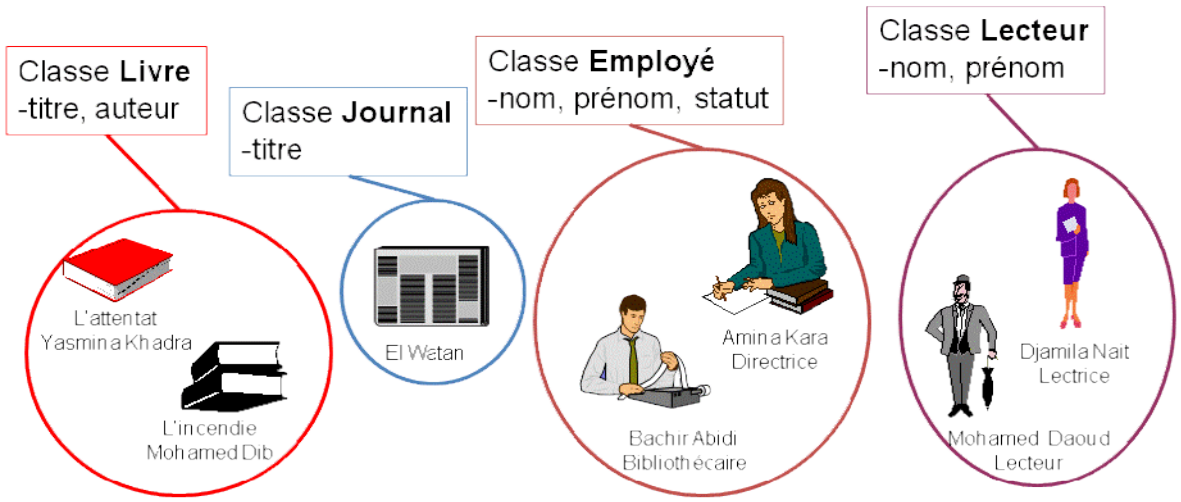
Objets



Classes

Des objets similaires peuvent être informatiquement décrits par une même abstraction: une **classe**

- même structure de données et méthodes de traitement
- valeurs différentes pour chaque objet



V.3.1 Déclaration

Une classe est une structure. Sauf que:

- Le mot réservé **class** remplace **struct**
- Certains champs sont des **fonctions**.

En C++, une classe se définit grâce au mot clef "**class**" suivi du nom de celle-ci. Ce nom commence généralement par une majuscule. A la suite, les données membres et les méthodes sont déclarées entre deux accolades. La définition de la classe se termine obligatoirement par un point virgule:

Syntaxe:

```
class Nom
{
   //données membres et fonctions membres
};
```

Prenons un exemple concret et simple: l'écriture d'une classe **Point**:

En **C**, nous aurions fait une structure comme suit:

```
struct Point
{
  int x; // Abscisse du point
  int y; // Ordonnée
};
```

La déclaration précédente fonctionne parfaitement en C++. Mais nous aimerions rajouter des fonctions qui sont fortement liées à ces données, comme l'affichage d'un point, son déplacement, etc.

Voici une solution en **C++**:

```
#include <iostream.h>
class Point
{  public : // voici les attributs
   int x;
   int y;
  // voici les méthodes
  void afficher()
  { cout <<x<<','<<y<<endl; }
```

```
  void placer (int a ,int b)
  {  x=a;
     y=b;
  }
  void deplace(int a, int b)
  {  x += a;
     y += b;
  }
};
```

Commentaires:

- Les champs x et y sont les **attributs** de classes, appelés aussi variables membres.
- Les fonctions **afficher**, **placer** et **deplace** sont appelées méthodes ou fonctions membres.
- Le terme **public** signifie que tous les membres qui suivent (données comme méthodes) sont accessibles de l'extérieur de la classe. Nous verrons les différentes possibilités plus tard.

V.3.2 Utilisation de la classe

V.3.2.1 Instance de classe

Une classe étant définie, il est possible de créer des objets (des variables ou des constantes) de ce type.
On les appelle alors des ***objets*** de la classe ou des ***instances*** de la classe on déclare un "objet" d'un type classe donné en faisant précéder son nom de celui de la classe, comme dans l'instruction suivante qui déclare deux objets a et b de type point.

Exemple :

```
point a, b ;
```

V.3.2.2 Accès au membre d'un objet

- ***Cas de déclaration statique:***

```
#include <iostream>
#include "Points.cpp"
void main()
{
  Point p;
  p.x=10;
  p.y=20;
  p.afficher();
  p.placer(1,5);
  p.afficher();
}
```

- ***Cas d'allocation Dynamique:***

```
#include <iostream>
#include "Points.cpp"
void main()
{
   Point *p=new Point;
```

```
 p->x=10;
 p->y=20;
 p->afficher();
 p->placer(1,5);
 p->afficher();
}
```

V.3.3 Opérateurs de résolution de portée

Les méthodes de la classe ***Point*** sont implémentées dans la classe même. Ceci fonctionne très bien, mais devient bien entendu assez lourd lorsque le code est plus long. C'est pourquoi il vaut mieux placer la déclaration seulement, au sein de la classe. Le code devient alors :

```
#include <iostream>
class Point
{ public :
  int x;
  int y;
  void placer(int a, int b);
  void deplace(int a, int b);
  void affiche();
};
void Point::placer(int a, int b)
{   x = a;
    y = b;
}
void Point::deplace(int a, int b)
{  x += a;
   y += b;
}
void Point::affiche()
{
  cout << x << ", " << y << endl;
}
void main()
{  Point p;
   p.placer (3,4);
   p.affiche();
   p.deplace(4,6);
   p.affiche();
}
```

✍ On remarque la présence du "**Point::**" (**:: opérateur de résolution de portée)** qui signifie que la fonction est en faite une méthode de la classe *Point*. Le reste est complètement identique.
La seule différence entre les 2 manières vient du fait qu'on dit que les fonctions de la première (dont l'implémentation est faite dans la classe), sont "**inline**". Ceci signifie que chaque appel à la méthode sera remplacé dans l'exécutable, par la méthode en elle-même. D'où un gain de temps certain, mais une augmentation de la taille du fichier en sortie.

V.3.4 Objets transmis en argument d'une fonction membre

Nous pouvons maintenant imaginer vouloir comparer deux points, afin de savoir s'ils sont égaux. Pour cela, nous allons mettre en œuvre une méthode "**Coincide**" qui renvoie "true" lorsque les coordonnées des deux points sont égales :

```
class Points
{  public :
   int x;
   int y;
   bool Coincide(Point p)
   {  if( (p.x==x) && (p.y==y) )
         return true;
     else
        return false;
   }
} ;
```

Cette partie de programme fonctionne parfaitement, mais elle possède un inconvénient majeur : le **passage de paramètre par valeur**, ce qui implique une "duplication" de l'objet d'origine. Cela n'est bien sûr pas très efficace.

La solution qui vous vient à l'esprit dans un premier temps est probablement de passer par un pointeur. Cette solution est possible, mais n'est pas la meilleure, dans la mesure où nous savons fort bien que ces pointeurs sont toujours sources d'erreurs (lorsqu'ils sont non initialisés, par exemple).

La vraie solution offerte par le C++ est de passer par des références. Avec ce type de passage de paramètre, aucune erreur est possible puisque l'objet à passer doit déjà exister (être instancier). En plus, les références offrent une simplification d'écriture, par rapport aux pointeurs :

```
#include <iostream>
class Points
{  public :
   int x;
   int y;
  bool Coincide(Point &p)
  { if( (p.x==x) && (p.y==y) )
     return true;
   else
     return false;
  }
};
void main()
{
Points p;
Points pp;
pp.x=0;pp.y=1;
p.x=0; p.y=1;
if( p.Coincide(pp) )
```

```
cout << "p et pp coincident !" << endl;
if( pp.Coincide(p) )
cout << "pp et p coincident !" << endl;
}
```

V.3.5 Fonction membre dont le type est le même que la classe:

Supposons qu'on a besoin de déterminer le point milieu entre deux points, donc on peut ajouter à la classe point une méthode qui permet de fournir ce point, cette méthode accepte comme argument un point P et elle doit produire le point milieu entre P et le point représenter par la classe d'où le résultat (type de retour) est un point.

Exemple:

```
class Points
{ public :
int x;
int y;
void init(int a, int b)
{ x=a;
  y=b;
}
Points Milieu(Point &p)
{  Point M;
   M.x=(x+P.x)/2 ;
   M.y=(y+P.y)/2 ;
   return M ;
}
};
void main()
{
Points P1, P2,M;
P1.init(1,3);
P2.init(4,4);
M=P1.Milieu(P2);
//ou bien M=P2.Milieu(P1);
}
```

V.3.6 Donnée membre de type structure (struct)

On veut définir une classe permettant de créer des 'personne'; une personne est caractérisée par son nom, prénom, numéro CIN et date de naissance.
Or la date de naissance est un champ composé par jour, mois et année, donc il faut le regroupé sous un seul nom a l'aide d'une structure Date:

1ère méthode : La structure est définie en dehors de la classe

```
typedef struct Date
{ int j,m,a;
} ;
class Etudiant
{ public :
   char Nom[50] ;
```

```
   char Prenom[50] ;
   int  NCIN ;
Date DN
//…
};
```

- La structure Date est connue dans tout le programme.

```
void main()
{  Etudiant E ;
   Date D ;
   D.j=1 ; D.m=1 ; D.a=2008 ;
   strcpy(E.Nom, "ali") ;
   E.DN=D ;
};
```

2ème méthode : La structure est définie à l'intérieur de la classe

```
class Etudiant
{ typedef struct Date
  { int j, m, a;  } ;
  public :
  char Nom[30] ;
  char Prenom[30] ;
  int  NCIN ;
  Date DN ;
//…
};
```

- La structure Date est connue uniquement à l'intérieur de la classe Etudiant.

```
void main()
{  Etudiant E ;
   Date D ; // erreur de compilation
   E.DN.j=1 ;
   D.DN.m=1 ;
   D.DN.a=2008 ;
   strcpy(E.Nom, "Nabil") ;
//…
};
```

V.3.7 Classe composée par d'autres classes:

Ici on parle **d'objets membres**: lorsqu'un attribut de la classe est lui même de type une autre classe.

Exemple:

```
class Date
{ public:
   int j, m, a ;
   void afficher()
    { cout<<j<< "/"<<m<<"/"<<a<<endl ;}
   void init(int jj, int mm, int aa)
    {  j=jj ;  m=mm ; a=aa ; }
};
```

```
class Etudiant
{   public :
     char Nom[30] ;
     char Prenom[30] ;
     int  NCIN ;
     Date DN ; // DN est une instance de Date
    void afficher()
    {
      cout<< "Nom="<<Nom<<endl ;
      cout<< "Prénom="<<Prenom<<endl ;
      cout<< "numéro de CIN="<<NCIN<<endl ;
      cout<< "Date de naissance= " ;
      DN.afficher() ;
    }
//...
};
```

- Déclaration de la classe Date dans Etudiant :

```
class Etudiant
{
 class Date
 {public :
    int j;
    int m;
    int a;
    void afficher()
    { cout<<j<< "/"<<m<<"/"<<a<<endl ;}
    void init(int jj, int mm, int aa)
    { j=jj ; m=mm ; a=aa ; }
  };
 public :
   char Nom[30] ;
   char Prenom[30] ;
   int NCIN ;
   Date DN ; // DN est une instance de Date
   void afficher()
   {  cout<< "Nom="<<Nom<<endl ;
      cout<< "Prénom="<<Prenom<<endl ;
      cout<< "numéro de CIN="<<NCIN<<endl ;
      cout<< "Date de naissance= " ;
      DN.afficher() ;
    }
//...
};
```

✍ Date est utilisable uniquement dans la classe Etudiant.

Chapitre VI: Notions d'Encapsulation / Constructeurs et Destructeurs

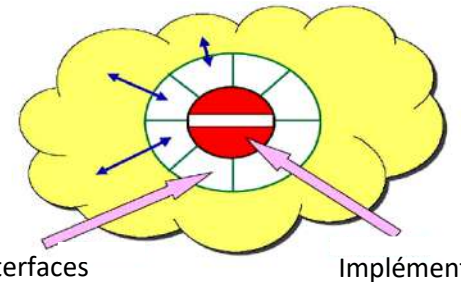
VI.1 Notion d'encapsulation

VI.1.1 Définition

L'encapsulation ou masquage d'information consiste à séparer les aspects externes d'un objet accessibles pour les autres, qu'on désigne par **publique**, des détails d'implémentation interne, rendus invisibles aux autres, qu'on désigne par **privé**.

- Technique qui isole l'aspect externe de l'objet de son aspect interne.
- Les données et les méthodes sont rassemblées ensemble. Elles sont cachées et protégées.

Ainsi **l'implémentation** d'un objet peut être modifiée sans affecter les **applications** qui emploient cet objet. Voici une figue caractéristique [11]:



Interfaces Implémentation

VI.1.2 Notion de visibilité

La visibilité d'une caractéristique détermine si d'autres classes peuvent l'utiliser directement. On utilise 3 niveaux de visibilité:

+ (Public): Les données et fonctions membres sont accessibles dans toute l'application.

 # (Protected): Les données et fonctions membres ne sont accessibles que par les fonctions membres de la classe et de toutes classes dérivées (voir héritage).

- (Private): Les données et fonctions membres ne sont accessibles que par les fonctions membres de la classe. Par défaut les **membres sont privés.**

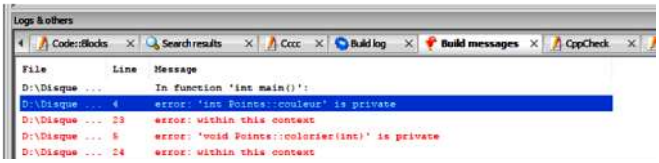
Exemple:

```
#include <iostream>
using namespace std ;
class Points
{  int couleur; //membre privé
   void colorier(int pcouleur)
   {  couleur = pcouleur; }
 public :
   int x, y;
   void afficher()
   {  cout <<this->x<<','<<this->y<<endl;} //Notation Inutile avec this
   void placer (int x , int y)
   {  this-> x = x;
      this->y = y;
   }
};
```

Programme principal:

```
#include <iostream>
#include "Points.cpp"
using namespace std ;
void main()
{   Points *p=new Points;
    Points *p1=new Points;
    p1->x=1;
    p1->y=15;
    p1->couleur=1;  // erreur de compilation car couleur est un membre privé
    p1->colorier(10);  // erreur de compilation car colorier() est une fonction privée
}
```

✍ **Messages d'erreurs:**



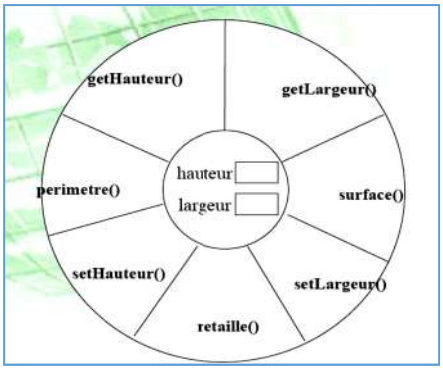
✍ **Remarque:** Les mots clés ***public***, ***private*** et ***protected*** peuvent apparaître à plusieurs reprises dans la déclaration d'une même classe.

VI.1.3 Méthodes d'accès et méthodes de modifications [7]

- **Les Accesseurs**: ***get<Nom Attribut>*** méthode de lecture d'un attribut privé.
- **Les Modificateurs(ou mutateurs)**: ***set<NomAttribut>*** méthode de modification d'un attribut.

Exemple:

```
class Rectangle
{public:
   int Perimetre()
   { return 2*(largeur+hauteur); }

   int surface()
   { return largeur*hauteur; }
////////////// Les ACCESSEURS
   int getLargeur()
   { return largeur; }

   int getHauteur()
   { return hauteur; }
////////////// Les MODIFICATEURS
   void setLargeur(int nouvelleLargeur)
   { largeur=nouvelleLargeur; }

   void setHauteur(int nouvelleHauteur)
   { hauteur=nouvelleHauteur; }
private :
   int largeur;
   int hauteur;
};
```



VI.2 Initialisation et constructeur

VI.2.1 Initialisation

L'initialisation permet d'affecter des valeurs à des variables lors de leur déclaration. Comment pouvons-nous initialiser un objet? Quel mécanisme offre la programmation orientée objet pour l'initialisation des objets.

Initialisation : Affectation de valeur lors de la déclaration

Exemple:

```
class compte
{ private:
   int numero; //numéro du compte
   float solde ;
  public :
   void init(int n)
   { numero=n ;
     solde=0.f ;
   }
   void consulter()
   {  cout <<"Le compte numero:"<<numero ;
      cout <<"\n a le solde : "<<solde ;
   }
};
```

Nous voulons initialiser un objet compte b :

- **1ère solution:**

```
compte b;
b.init(100);
```

  - Correcte mais nous n'avons pas fait d'initialisation, en effet, l'affectation est faite après la déclaration. En plus nous pouvons avoir des comptes qui sont manipulés sans avoir un numéro de compte parce que nous pouvons déclarer des objets compte sans être obligé d'appeler init.

- **2ème solution:**

```
compte b={100,0.f} ;
```

  - Correcte mais numéro et solde sont privés donc nous ne pouvons pas les accéder. En plus même si dans une classe tous ses attributs sont publiques, il faut tous les initialiser et dans l'ordre de leur apparition dans la classe, ce qui n'est pas aisée.

⇒ Vous remarquez donc que pour initialiser un objet nous avons besoin d'un nouveau mécanisme: **Constructeur.**

VI.2.2 Constructeurs

Un constructeur est une fonction membre spéciale dont la tâche est d'initialiser l'objet; elle porte le même nom que la classe, elle n'a pas de résultat, et elle peut avoir 0 ou plusieurs paramètres.
Elle est appelée lors de la déclaration de l'objet d'une façon implicite.

Exemple :

```
class Cpoint
{   float abscisse;
    float ordonne;

  public:
    Cpoint (float x =0.f, float y =0.f)
    { abscisse = x;
      Ordonne = y;
    }
};
void main()
{    Cpoint p1(5,3);  //p1 a les coordonnées (5,3)
     Cpoint p2(15);  //p2 a les coordonnées (15,0)
     Cpoint p3;      //p3 a les coordonnées (0,0)
}
```

VI.2.3. Quelques règles sur les constructeurs

Un constructeur doit s'en tenir aux règles suivantes:

- Un constructeur doit porter le même nom que sa classe.
- Un constructeur ne peut pas être défini avec une valeur de retour même pas void.
- Un constructeur sans paramètres est un constructeur par défaut.
- Un constructeur dont tous les paramètres ont des paramètres par défaut est un constructeur par défaut.
- Un constructeur doit être public.

VI.2.4 Surcharge des noms de constructeurs [7]

Une classe peut comporter autant de constructeurs que nécessaire.

Exemple:

```
class compte
{   int numero; //numéro du compte
     float solde ;
 public :
   compte(int n) ;
   compte(int n,float montant) ;
   compte() ;
   void consulter() ;
   void debiter(float montant) ;
   void crediter(float montant) ;
};
void main()
{ compte b (1000);         // numéro=1000 et solde=0
  compte c(1500,200);  //numéro =1500 et solde=200
}
```

VI.2.5 Constructeur et tableau d'objet:

Si nous déclarons un tableau d'objets, le constructeur par défaut sera appelé autant de fois que la taille du tableau.

Exemple:

```
compte C[5];    //le constructeur sera appelé 5 fois
compte *pc=new compte[10];    //le constructeur sera appelé 10 fois
```

VI.3 Constructeur par recopie

Reprenons notre classe *Point*. Il apparaît qu'il pourrait être intéressant de réaliser un constructeur à partir d'un point! C'est le constructeur **par recopie**: c'est un constructeur ayant un seul paramètre objet de la même classe. Ce constructeur initialise un objet à partir du contenu des attributs du paramètre [7].

Exemple:

```
class Point       // Définition de la classe Point
{   int x;
    int y;
  public :
   Point()
    { x = -1;  Y = -1;  }
   Point(int a, int b)
   {  x = a;  y = b;   }
//Constructeur par recopie
   Point(const Point &pt)
    {  x = pt.x;   y = pt.y;
    }
... // D'éventuelles autres méthodes
};
...
Point pt(1,2);
Point pt2(pt);   //Construction par recopie de Point
...
```

⚠ C++ oblige un passage par référence dans le cas d'un constructeur par recopie. Ceci vient du fait qu'il faut réellement utiliser l'objet lui-même, et non une copie.

Dans le cas d'un objet qui comporte un pointeur, lorsqu'on veut recopier un objet, on ne recopie pas ce qui est pointé, mais l'adresse du pointeur seulement. Du coup, on se retrouve avec deux objets différents, mais qui possèdent une donnée qui pointe vers la même chose !

Exemple:

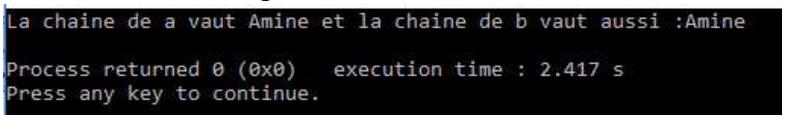
```
#include <string.h>
class PointNomme
{ public:
    PointNomme(int a, int b, char *s="")
    {   x = a;
        y = b;
        label=new char[strlen(s)+1];
        strcpy(label, s);
    }

   int x, y;
   char *label;
};
```

Regardons le Programme principal:

```
#include <iostream>
#include "PointNomme.cpp"
using namespace std ;
void main()
{
  PointNomme *a= new PointNomme(1,1);
  PointNomme *b=a;     //a et b partagent les mêmes valeurs!
  b->label="Bonjour";    //regardez cette affectation!
  a->label="Amine";
  cout <<"La chaine de a est changé mais la chaine de b vaut aussi :"<<b->label<<endl;
}
```

✍ On remarque que nous avons effectué l'affectation avant de changer l'attribut **label** de a et pourtant **b->label** a aussi changé!

Exécution :

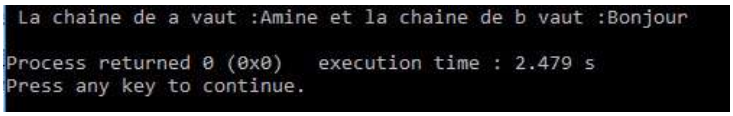


Cependant si on cherche à faire la duplication de a dans le but de gérer séparément les objets « égaux » pointés par a et b:

```
#include <string.h>
class PointNomme
{  public:
     PointNomme(int a, int b, char *s="")
     { x=a; y=b;
       label=new char[strlen(s)+1];
       strcpy(label,s);
     }
    PointNomme(const PointNomme &p) // Constructeur à partir d'un objet existant!
    { x=p.x;
      y=p.y;
      label=new char[strlen(p.label)+1];
      strcpy(label,p.label);
    }
   int x,y;
   char *label;
};
```

```
#include <iostream>
#include "PointNomme.cpp"
using namespace std ;

void main()
{  PointNomme *a= new PointNomme(1,1);
  PointNomme *b=new PointNomme(*a);
  b->label="Bonjour";
  a->label="Amine";
 cout<<" La chaine de a vaut :"<<a->label<<" et la chaine de b vaut :"<<b->label <<endl;
}
```

Résultat de l'exécution:



VI.4 Construction des objets membres

Ici on parle **d'objets membres**: lorsqu'un attribut de la classe est lui même de type une autre classe. L'initialisation d'un objet de la classe nécessite alors l'initialisation de ses objets membres. Si les objets membres n'ont que des constructeurs par défaut, le problème ne se pose pas!

Sinon voici la syntaxe :

```
NomClasse(paramètres):membre1(paramètres),membre2(paramètres),…
    {
     Corps du constructeur de NomDeLaClasse
    }
```

Voici un exemple : On considère la classe **Point** et la classe **Segment** formée par deux objets membres origine de classe Point et **extremite** de classe Point. L'appel des constructeurs s'effectue de la façon suivante :

```
class Point
{   public:
     Point(int px,int py)
     { x=px; y=py; }
  private:
     int x;
     int y;
};
class Segment
{    Point origine; //Objet membre
    Point extremite; //Objet membre
    int epaisseur;
 public:
    Segment(int ox,int oy,int ex,int ey,int ep): origine(ox,oy),extremite(ex,ey)
    { epaisseur=ep; }
};
```

IV.5 Destructeurs

Tout comme il existe un constructeur, on peut spécifier un destructeur. Ce dernier est appelé lors de la destruction de l'objet, explicite ou non.

Un destructeur d'une classe donnée est une méthode exécutée **automatiquement** à chaque fois qu'une instance de la classe donnée **disparait**.

L'identificateur est celui de la classe précédé du caractère **~** :

- C'est une méthode sans type de retour ;
- C'est une méthode sans paramètre, elle **ne peut donc pas être surchargée** ;
- C'est une méthode en accès **public**.

Exemple:

```
class Point
 { double x, y ;
   int norm ;
 public :
  // Les constructeurs
   Point(double xx, double yy);
   Point(int n);
 // Le destructeur
   ~Point();
};
```

✍ Grâce à la surcharge des noms de fonctions, il peut y avoir plusieurs constructeurs, mais il ne peut y avoir qu'un seul destructeur.

Le destructeur d'une classe C est appelé **implicitement**:

- A la disparition de chaque objet de classe C.
- A la fin du main() pour toutes les variables statiques, locales au main() et les variables globales.
- A la fin du bloc dans lequel la variable automatique C est déclarée.
- A la fin d'une fonction ayant un argument de classe C.
- Lorsqu'une instance de classe C est détruite par **DELETE**.
- Lorsqu'un objet qui contient un attribut de type classe C est détruit.
- Lorsqu'un objet d'une classe dérivée de C est détruit.

Exemple :

Le programme suivant illustre quelques cas:

```
#include <iostream>
using namespace std ;
class test
{   int attr;
  public:
    test()
    { cout<<"----- contructeur par defaut!"<<endl; }

    ~test()
    { cout<<"----- Le destructeur ! "<<endl; }
};

void blocLocal()
{  cout<<"BLOC LOCAL : "<<endl;    test CTlocal; }
void main()
{  cout<<"BLOC MAIN : "<<endl;
   test *CTmain = new test;
   cout<<"Appel du bloc local dans main : "<<endl;
   blocLocal ();
   cout<<"APPEL DE DELETE : "<<endl;
   delete CTmain;
}
```

Résultat de l'exécution :

```
BLOC MAIN :
----- contructeur par defaut!
Appel du bloc local dans main :
BLOC LOCAL :
----- contructeur par defaut!
----- Le destructeur !
APPEL DE DELETE :
----- Le destructeur !
```

Chapitre VII: Patrons et amies " Fonctions et Classes "

VII.1 Les patrons « Template »

Parmi les techniques pour améliorer la réutilisabilité des morceaux de code, nous trouvons la notion de généricité. Cette notion permet d'écrire du code générique en paramétrant des fonctions et des classes par un type de données. Un module générique n'est alors pas directement utilisable : c'est plutôt un modèle, patron (***template***) de module qui sera «instancié » par les types de paramètres qu'il accepte. Dans la suite nous allons montrer comment C++ permet, grâce à la notion de ***patron de fonctions***, de définir une ***famille de fonctions paramétrées*** par un ou plusieurs types, et éventuellement des expressions. D'une manière comparable, C++ permet de définir des "***patrons de classes***". Là encore, il suffira d'écrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter à différents types.

VII.1.1 Patrons de fonctions

Pour illustrer les ***patrons de fonctions***, prenons un exemple concret : une fonction min qui accepte deux paramètres et qui renvoie la plus petite des deux valeurs qui lui est fournie. On désire bénéficier de cette fonction pour certains types simples disponibles en C++ (int, char, float). Les notions que nous avons vu en C++ jusqu'à maintenant ne nous permettent de résoudre ce problème qu'avec une seule solution. Cette solution est d'utiliser la surcharge et de définir trois fonctions min, une pour chacun des types considérés.

Exemple :

```
int min (int a, int b)
{ if ( a < b)
   return a ; // return ((a < b)? a : b);
  else
   return b ;
}

float min (float a, float b)
{ if ( a < b)
    return a ;
  else
     return b ;
}

char min (char a, char b)
{  if ( a < b)
    return a ;
  else
    return b ;
}
```

✍ Définition des fonctions min grâce à la surcharge: lors d'un appel à la fonction min, le type des paramètres est alors considéré et l'implantation correspondante est finalement appelée. Ceci présente cependant quelques inconvénients :

- La définition des 3 fonctions (perte de temps, source d'erreur) mène à des instructions identiques, qui ne sont différenciées que par le type des variables qu'elles manipulent.
- Si on souhaite étendre la définition de cette fonction à de nouveaux types, il faut définir une nouvelle implantation de la fonction min par type considéré.

✍ Une autre solution est de définir une fonction ***template***, c'est-à-dire générique. Cette définition définit en fait un patron de fonction, qui est instancié par un type de données (ici le type T) pour produire une fonction par type manipulé.

Exemple :

```
template <class T> // T est le paramètre de modèle
T min (T a, T b)
{  if ( a < b)
     return a
   else
     return b;
}
void main(){
int a = min(1, 7);  // int min(int, int)
float b = min(10.0, 25.0);    // float min(float, float)
char c = min('z', 'c');    // char min(char, char)
}
```

✍ Définition de la fonction min générique : il n'est donc plus nécessaire de définir une implantation par type de données. On définit donc bien plus qu'une fonction, on définit une méthode permettant d'obtenir une certaine abstraction en s'affranchissant des problèmes de type.

Remarques :

- Il est possible de définir des ***fonctions template*** acceptant plusieurs types de données en paramètre. Chaque paramètre désignant une classe est alors précédé du mot-clé ***class***, comme dans l'exemple : ***template <class** T, **class** U**> ....***
- Chaque type de données paramètre d'une ***fonction template*** doit être utilisé dans la définition de cette fonction.
- Pour que cette fonctionnalité soit disponible, les fonctions génériques doivent être définies au début du programme ou dans des fichiers d'interface (fichiers .h).

VII.1.2 Classe template : patron de classes

Il est possible, comme pour les fonctions, de définir des ***classes template***, c'est-à-dire paramétrées par un type de données. Cette technique évite ainsi de définir plusieurs classes similaires pour décrire un même concept appliqué à plusieurs types de données différents. Elle est largement utilisée pour définir tous les types de containers (comme les listes, les tables, les piles, etc.), mais aussi des algorithmes génériques par exemple.

La syntaxe permettant de définir une ***classe template*** est similaire à celle qui permet de définir des ***fonctions template***.

Exemple :

```
int x, y ;
public :
point (int abs=0, int ord=0) ;
void affiche () ;
// .....
} ;
```

✍ Lorsque nous procédons ainsi, nous imposons que les coordonnées d'un point soient de valeurs de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type (float, double, long ...), nous devons définir une autre classe en remplaçant simplement, dans la classe précédente, le mot clé int par le nom de type voulu.

Ici encore, nous pouvons simplifier considérablement les choses en définissant un seul patron de classe de cette façon:

```
template <class T> class point {
T x , y ;
public :
point (T abs=0, T ord=0) ;
void affiche () ;
} ;
```

✍ Comme dans le cas des patrons de fonctions, la mention ***template <class T>*** précise que l'on a affaire à un patron (template) dans lequel apparaît un paramètre de type nommé T ;
La définition de notre patron de classes n'est pas encore complète puisqu'il y manque la définition des fonctions membres, à savoir le constructeur point et la fonction affiche().

Pour ce faire, la démarche va légèrement différer selon que la fonction concernée est en ligne ou non. Voici par exemple comment pourrait être défini notre constructeur en ligne:

```
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
```

En revanche, lorsque la fonction est définie en dehors de la définition de la classe, il est nécessaire de rappeler au compilateur :

- que, dans la définition de cette fonction, vont apparaître des paramètres de type ; pour ce faire, on fournira à nouveau la liste de paramètre sous la forme: ***template <class T>***
- le nom du patron concerné. Par exemple, si nous définissons ainsi la fonction affiche, son nom sera: ***point<T>::affiche ()***

En définitive, voici comment se présenterait l'en-tête de la fonction affiche si nous le définissions ainsi en dehors de la classe :

```
template <class T>
void point<T>::affiche ()
```

En toute rigueur, le rappel du paramètre T à la suite du nom de patron (point) est redondant puisqu'il a déjà été spécifié dans la liste de paramètres suivant le mot clé template.

Voici ce que pourrait être finalement la définition de notre patron de classe point :

```
template <class T> class point    // Création d'un patron de classe
{  T x , y ;
  public :
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{ cout << "Paire : " << x << " " << y << "\n" ; }
```

VII.1.3 Utilisation d'un patron de classes

Comme pour les patrons de fonctions, l'instanciation de tels patrons est effectuée automatiquement par le compilateur selon les déclarations rencontrées. Après avoir créé ce patron, une déclaration telle que:

point <int> *a;*

conduit le compilateur à instancier la définition d'une classe point dans laquelle le paramètre T prend la valeur ***int***. Autrement dit, tout se passe comme si nous avions fourni une définition complète de cette classe. Si nous déclarons :

point <float> ad ;

le compilateur instancie la définition d'une classe point dans laquelle le paramètre T prend la valeur **float**, exactement comme si nous avions fourni une autre définition complète de cette classe.
Si nous avons besoin de fournir des arguments au constructeur, nous procéderons de façon classique comme dans :

point <int> a (3, 5) ;
point <float> ad (1.5, 9.6) ;

Si nous faisons abstraction de la signification des paramètres (coordonnées d'un point) et nous les déclarons comme des caractères ou chaînes de caractères, le compilateur ne signalera pas d'incohérence et la fonction *affiche()* remplacera *x* et *y* par les arguments, comme le montre l'exemple récapitulatif.

Exemple récapitulatif :

Voici un programme complet comportant :

- la création d'un patron de classes point dotée d'un constructeur en ligne et d'une fonction membre (affiche) non en ligne,
- la création d'un patron de fonctions min (en ligne dans la patron de classes) qui retourne le minimum des arguments,
- un exemple d'utilisation (main).

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

template <class T> class point   // Création d'un patron de classe
{  T x ;
   T y ;
public :
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
void affiche () ;

T min ()
{ if ( x < y)
    return x;
  else
    return y;
}
};

template <class T> void point<T>::affiche ()
{ cout << "Paire : " << x << " " << y << "\n" ;  }

void main ()
{  point <int> ai (3, 5) ; // T prend la valeur int pour la classe point
    ai.affiche() ;
    cout << " Min : ... " << ai.min() << endl;
    point <char> ac ('z', 't') ; ac.affiche() ;
    cout << " Min : ... " << ac.min() << endl;
    point <double> ad (1.5, 9.6) ; ad.affiche() ;
    cout << " Min : ... " << ad.min() << endl;
    point <string> as ("Salut", " A vous") ; as.affiche() ;
    cout << " Min : ... " << as.min() << endl;
}
```

Résultat de l'exécution:

```
Paire : 3 5
 Min : ... 3
Paire : z t
 Min : ... t
Paire : 1.5 9.6
 Min : ... 1.5
Paire : Salut  A vous
 Min : ...  A vous

Process returned 0 (0x0)   execution time : 3.488 s
Press any key to continue.
```

Remarque :

Il faut noter aussi que le nombre de paramètres n'est pas limité à 1, on peut en avoir plusieurs :

```
template < class A, class B, class C, ... >
class MaClasse
{
// ...
public:
// ...
};
```

L'exemple suivant montre comment implémenter les fonctions de la classe template.

```
template <class T, class T1, class T2> class Etudiant
{  T Nom, Prenom;
   T1 Id;
   T2 Age;
public:
//...
T getNom()
 {return nom;}
T1 getId()
 {return Id;}
T2 getAge();
void Saisie();
void affiche();
//...
};

template <class T, class T1, class T2> T2 Etudiant<T,T1,T2>::getAge()
{ return Age; }

template <class T, class T1, class T2> void Etudiant<T,T1,T2>::affiche ()
{  cout << "Data : " << Nom << " " << Id << " " << Age <<"\n" ; }
```

VII.2 Fonctions amies

VII.2.1 Problème

Comment faire pour qu'une ***fonction f()*** et/ou une ***classe B*** puisse accéder aux données membres privées d'une ***classe A*** ?

Exemple :

```
class ratio
{  private:
     int num, den ;
   public:
    ratio(int n, int d);
    void affiche();
    float val_reel();
}
ratio somme (ratio r1, ratio r2);
```

Solution:[2]

- **Rendre publiques les données membres des classes.**
  Inconvénient: on perd leurs protections.
- **Ajout de fonctions d'accès aux membres privés.**
  Inconvénient : temps d'exécution pénalisant.
- **Fonctions amies :** Il est possible de déclarer qu'une ou plusieurs fonctions (extérieurs à la classe), sont des « amies » ; une telle déclaration d'amitié les autorise alors à accéder aux données privées au même titre que n'importe quelle fonction membre. L'avantage de cette méthode est de permettre le contrôle des accès au niveau de la classe concernée.

Exemple de déclaration d'une fonction amie:

```
class A
{   private :
     int i ;
     friend class B ;
     friend void f();
}
class B
{
   //tout ce qui appartient à B,
  //peut se servir des données membres privés de A
 // comme si elle était de A.
...
}
void f(A& a)
{  a.i = 10 ;  }
```

VII.2.2 Situation d'amitiés

- Fonction indépendante amie d'une classe.
- Fonction membre d'une classe amie d'une autre classe.
- Fonction amie de plusieurs classes
- Toutes les fonctions membres d'une classe, amies d'une autre classe.

VII.2.2.1 Exemple de fonction indépendante amie d'une classe [2]

```
class point
{  private :
     int x,y ;
   public :
     point (int abs= 0, int ord = 0)
     { x= abs ;
       y= ord ;
     } ;
     friend int coincide (point p, point q) ;  // déclaration d'une fonction amie indépendante
} ;
int coincide (point p, point q)
{  if ((p.x == q.x) && (p.y == q.y) )
```

```
    return (1);
  else
    return (0);
} ;
void main()
{  point a(1,0), b (1), c ;
  if ( coincide (a,b) )
    cout<< "A coincide avec B"<<endl;
  else
     cout<<"A et B sont différents"<<endl;
} ;
```

Résultat de l'exécution :

```
A coincide avec B

Process returned 0 (0x0)   execution time : 2.785 s
Press any key to continue.
```

VII.2.2.2 Fonction membre d'une classe amie d'une autre classe

On considère deux classes A et B

int f(char, A) fonction membre de B

f doit pouvoir accéder aux membres privés de A, elle sera déclarée amie au sein de la classe A :

friend int B :: f(**char**, A) ;

Exemple :

```
class A
{  private :
   // partie privée
  public :
   // partie publique
  friend int B :: f (char, A) ;
  ...
  ...
} ;
class B
{  private :
    // partie privée
  public :
   // partie publique
   int f (char, A) ;
   ...
   ...
} ;
int B :: f (char, A) ;
{   // on a ici accès aux membres privés de tout objet de type A
} ;
```

VII.2.2.3 Fonction amie de plusieurs classes

Rien n'empêche qu'une même fonction (indépendante ou membre) fasse l'objet de déclaration d'amitié dans différentes classes.

✍ Toutes les fonctions d'une classe sont amies d'une autre classe

```
friend class B ; // dans la classe A
```

VII.2.4 Toutes les fonctions membres d'une classe, amie d'une autre classe

C'est une généralisation du cas précédent. On pourrait d'ailleurs effectuer autant de déclarations d'amitié qu'il n'y a de fonctions concernées. Mais il est plus simple d'effectuer une déclaration globale. Ainsi pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on placera, dans la classe A, la déclaration [2] :

```
friend class B ;
```

Exemple :

```
class A
{  // partie privée
   ......
  // partie publique
  friend class B;
  ......
};
class B
{   ......
    //on a accès totale aux membres privés de tout
   //objet de type A
    ......
};
```

Chapitre VIII: Surcharge d'opérateurs

VIII.1 Introduction : surcharge d'opérateurs

C++ autorise la surdéfinition de fonctions, qu'il s'agisse de fonctions membres ou de fonctions indépendantes.

Exemple:

```
class ratio
{  private :
    int num, den ;
  public :
     ratio ( n=0, d=1) ;
     ratio(n) ;
} ;
```

Attribuer le même nom à des fonctions différentes, lors de l'appel, le compilateur fait le choix de la bonne fonction suivant le nombre et les types d'arguments.

C++ permet aussi la surdéfinition d'opérateurs [2]

Exemple :

a+b ;

Le symbole + peut designer suivant le type de a et b :

- L'addition de deux entiers
- Addition de deux réels
- Addition de deux doubles.
- + est interprété selon le contexte.

En C++ on peut surdéfinir n'importe quel opérateur existant (unaire ou binaire). Ce qui va nous permettre de créer par le biais de classes, des types avec des opérateurs parfaitement intégrés.

✍ On peut donner une signification à des expressions comme a+b, a-b, a*b et a/b pour la classe ratio

VIII.2 Mécanisme de surdéfinition [2]

Considérons la classe point :

```
class point
{  private :
    int x, y ;
  public :
    // partie publique
    ...
} ;
```

Supposons que nous souhaitons définir l'opérateur (+) afin de donner une signification à l'expression a+b, tels que, a et b sont de type point.

On considère que la somme de deux points est un point dont les coordonnées sont la somme de leurs coordonnées.

Pour ***surdéfinir*** (***surcharger***) cet opérateur en C++, il faut définir une fonction de nom :

Operatorxx où **xx** est le symbole de l'opérateur.

La fonction **operator+** doit disposer de deux arguments de types point et fournir une valeur de retour de même type.

Cette fonction peut être définie en tant que fonction membre de la classe (méthode) ou fonction indépendante (fonction amie).

VIII.3 Surcharge d'opérateurs par des fonctions amies

Le prototype de notre fonction sera : **point operator+ (point, point)**
Les deux opérandes correspondent aux deux opérandes de l'opérateur +
Le reste du travail est classique [2] :

- Déclaration d'amitié
- Définition de la fonction

Exemple :

```
class point
{  private :
    int x, y ;
  public :
    point (int abs =0, ord =0) ;
    friend point operator+ (point , point) ;
    void afficher( );
} ;
point operator+ (point a, point b)
{
point p;
p.x =a.x +b.x;
p.y = a.y + b.y;
return p;
}
void main()
{
 point a (1,2);
 point b (2,5);
 point c;
 c= a+b; c.afficher();
 c= a+b+c; c.afficher();
}
```

On a surchargé l'opérateur **+** pour des objets de type point en employant une fonction amie.

VIII.4 Surcharge d'opérateurs par des fonctions membres

Dans ce cas, le premier opérande sera l'objet ayant appelé la fonction membre

Exemple :

L'expression a+b sera interprétée par le compilateur comme a.operator+ (b) ;
Le prototype de notre fonction membre est alors: point operator+ (point);

Exemple:

```
class point
{  private:
    int abs, ord;
  public:
    point (int abs, int ord);
    point operator+ (point a);
    void affiche();
};
point point ::operator+ (point a);
{  point p;
  p.x = x + a.x;
  p.y = y + a.y;
  return (p) ;
}
void main()
{
  point a (1,2);
  point b (2,5);
  point c;
  c= a+b; c.afficher();
  c= a+b+c; c.afficher();
}
```

Remarque :

Dans ce cas: c = a+b; c = a.operator+ (b);

Dans le 1er cas : a= operator+(a,b);

VIII.5 Surcharge en général

On a vu un exemple de surdéfinition de l'opérateur + lorsqu'il reçoit deux opérandes de type point. Ces de deux façons :

- fonctions amies
- fonctions membres

Remarques :

- Il faut se limiter aux opérateurs existants. Le symbole qui suit le mot clé **operator** doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est pas permis de créer de nouveaux symboles.
- Certains opérateurs ne peuvent pas être redéfinis de tout (.)
- Il faut conserver la pluralité de l'opérateur. unaire ou binaire.

  Binaire : + - / * -> ( )

  Unaire : -- ++ new delete
- Il faut se placer dans un contexte de classe :

On ne peut surdéfinir un opérateur que s'il comporte au moins un argument de type classe → Une fonction membre.

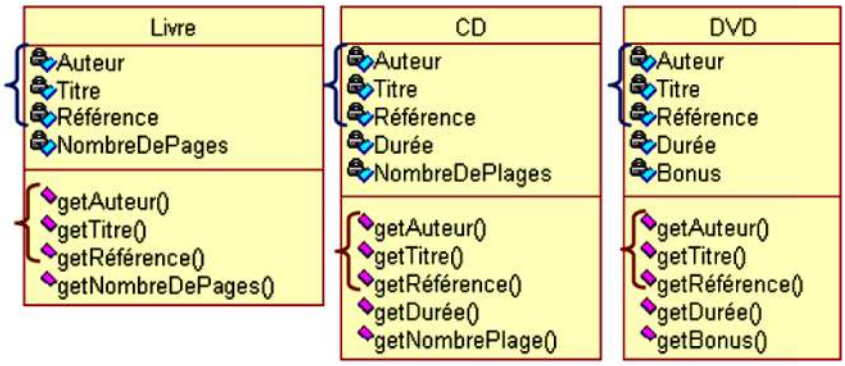
Une fonction indépendante ayant au moins un argument de type classe (Fonction amie).

Chapitre IX: HERITAGE EN C++

IX.1 Introduction

IX.1.1 Exemple [7]

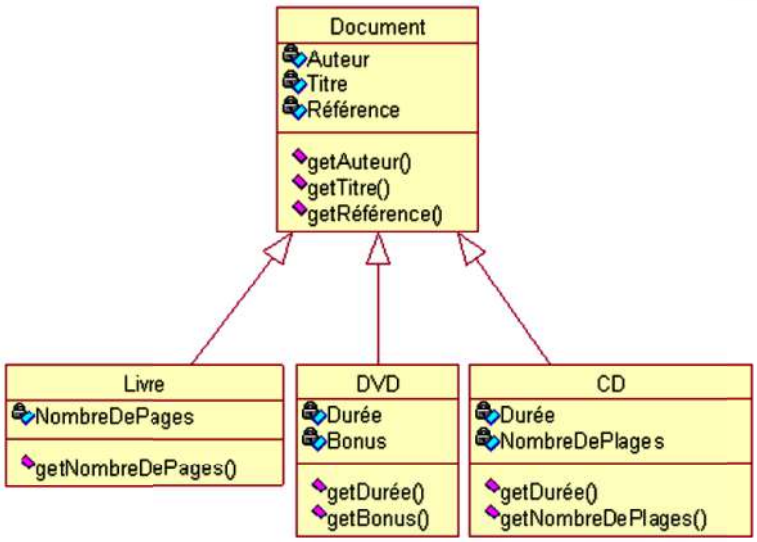
Imaginons que nous devions fabriquer un logiciel qui permet de gérer une bibliothèque. Cette bibliothèque comporte plusieurs types de documents; des livres, des CDs, ou des DVDs. Une première étude nous amène à mettre en œuvre les classes suivantes:



✍ Nous remarquons que dans les trois types de documents, un certain nombre de caractéristiques se retrouvent systématiquement.

Afin d'éviter la répétition des éléments constituant chacune des classes, il est préférable de ***factoriser toutes ces caractéristiques communes*** pour en faire une nouvelle classe plus généraliste.

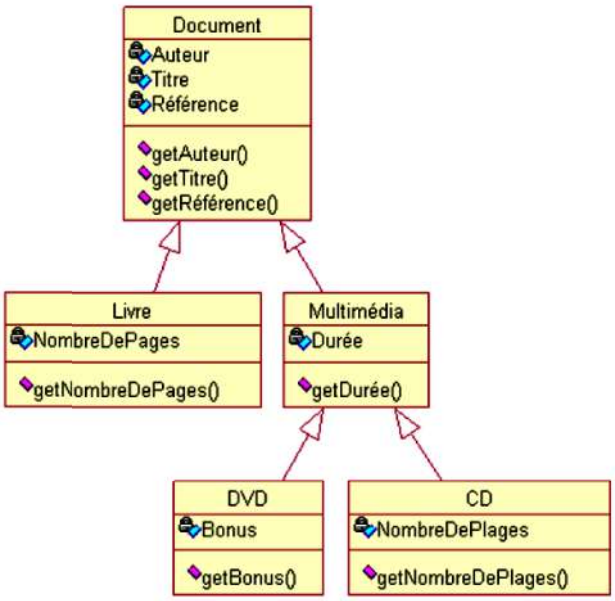
En effet, nous pouvons dire que, d'une façon générale, et quelque soit le type de document, il comporte au moins un titre, un auteur, etc. il semble aller de soi, que le nom de cette nouvelle classe générale s'appelle justement Document.



La généralisation se représente par une flèche qui part de la classe fille vers la classe mère. Par exemple, Un Livre possède, certes un nombre de page, mais en suivant la flèche indiquée par la relation de généralisation, elle comporte également un nom d'auteur, un titre, une

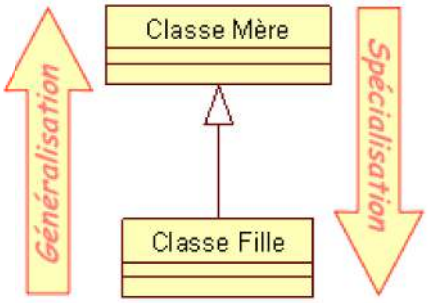
référence. En fait, la classe Livre hérite de tout ce que possède la classe Document, les attributs comme les méthodes.

Dans cet exemple, nous avons un seul niveau d'héritage, mais il est bien entendu possible d'avoir une hiérarchie beaucoup plus développée. D'ailleurs, si nous regardons de plus prêt, nous remarquons que nous pouvons appliquer une nouvelle fois la généralisation en factorisant la durée du support CD et du support DVD. En fait, il s'agit dans les deux cas d'un support commun appelé Multimédia.



IX.1.2 Définitions

- *L'héritage* est une technique permettant de construire une classe à partir d'une ou de plusieurs autres classes dites : *classe mère* ou *superclasse* ou *classe de base.*
- La classe dérivée est appelée: *Classe fille* ou *sous-classe.*



- Les sous-classes héritent des caractéristiques de leurs classes parents.
- Les attributs et les méthodes déclarées dans la classe mère sont accessibles dans les classes fils comme s'ils avaient été déclarés localement.

✍ Créer facilement de nouvelles classes à partir de classes existantes (réutilisation du code)

IX.2 Héritage simple

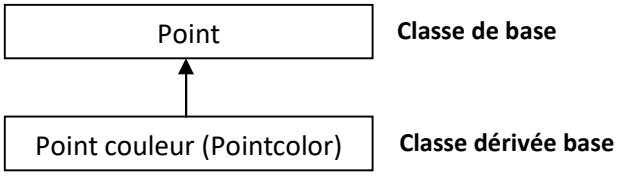
IX.2.1 Définition: *La classe dérivée hérite les attributs et les méthodes d'une seule classe mère.*

Syntaxe:

```
class ClasseMere
{  //…
};
class ClasseDerivee: <mode> ClasseMere
{  //…
};
```

✍ ***mode:*** optionnel permet d'indiquer la nature de l'héritage: *private*, *protected* ou *public. S*i aucun mode n'est indiqué alors l'héritage est privé.

Exemple :




```
class Point
{   int x;
    int y;
 public:
    void initialise (int , int) ;
    void afficher() ;
};

class Pointcolor: public Point
{   int couleur;
  public:
    void setcolor(int c)
    {  couleur=c ;  }
};

void main()
{   Pointcolor p ;
   p.initialiser (10,20) ;
   p.setcol(5) ;
   p.afficher() ;   // (10, 20)
}
```

IX.2.2 Utilisation des membres de la classe de base dans une classe dérivée

L'exemple précédent, destiné à montrer comment s'exprime l'héritage en C++, ne cherchait pas à explorer toutes les possibilités.

Or la classe *pointcolor* telle que nous l'avons définie présente des lacunes. Par exemple, lorsque nous appelons *affiche* pour un objet de type *pointcolor* nous n'obtenons aucune information sur sa couleur.

Une première façon d'améliorer cette situation consiste à écrire une nouvelle fonction membre public :

```
void Affichercolor()
{   cout << "("<<x<<","<<y<<")"<<endl ;
    cout << "couleur="<< couleur<<endl ;
}
```

Mais alors cela signifierait que la fonction *Affichercolor* membre de la classe *Pointcolor* aurait un accès aux membres privés de *point* ce qui serait contraire au principe d'encapsulation.

En revanche, rien n'empêche à une classe dérivée d'accéder à n'importe quel membre public de sa classe de base. D'ou une définition possible d' *Affichercolor:*

```
void Affichercolor()
{   afficher() ;
   cout << "couleur="<< couleur<<endl ;
}
```

D'une manière analogue, nous pouvons définir dans *pointcolor* une fonction d'initialisation comme *initialisercolor :*

```
void initialisercolor(int a, int b, int c)
{   initialiser(a,b) ;
   couleur=c ;
}
```

IX.2.4 Contrôle d'accès pendant l'héritage

Statut des membres de la classe dérivée en fonction du statut des membres de la classe de base et du mode de dérivation [7]:

		Statut des membres de base		
		Public	**Protected**	**Private**
Mode de dérivation	**Public**	Public	Protected	Inaccessible
	Protected	Protected	Protected	Inaccessible
	Private	Private	Private	Inaccessible

Remarque :

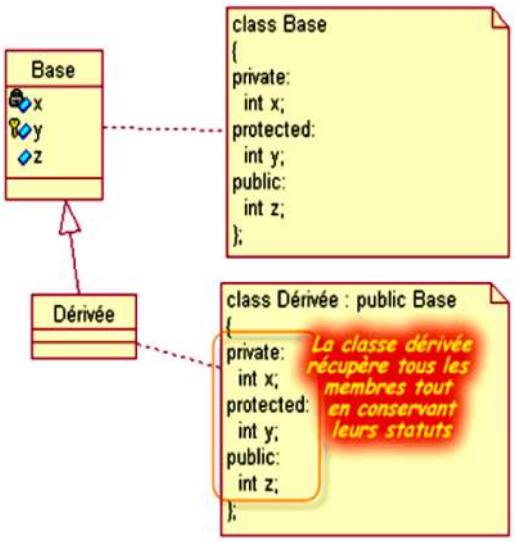
- Lorsqu'une classe dérivée possède des fonctions amies, ces derniers disposent exactement des mêmes autorisations d'accès que les fonctions membres de la classe dérivée. En particulier, les fonctions amies d'une classe dérivée auront bien accès aux membres déclarés protégés dans sa classe de base.
- En revanche, les déclarations d'amitié ne s'héritent pas. Ainsi, si *f* a été déclaré amie d'une classe A et si B dérive de A, *f* n'est pas automatiquement amie de B.

Dérivation publique [14]**:**

1. Les membres *publics* de la classe de base sont accessibles « à tout le monde », c'est-à dire à la fois aux méthodes, aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.

2. Les membres *protégés* de la classe de base sont accessibles aux méthodes et aux fonctions amies de la classe dérivée, mais pas aux utilisateurs de cette classe dérivée.

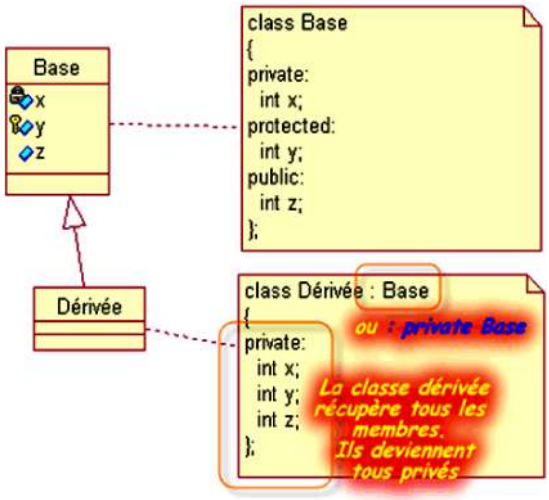
3. Les membres *privés* de la classe de base sont inaccessibles à la fois aux méthodes ou aux fonctions amies de la classe dérivée et aux utilisateurs de la classe dérivée.



Dérivation privée:

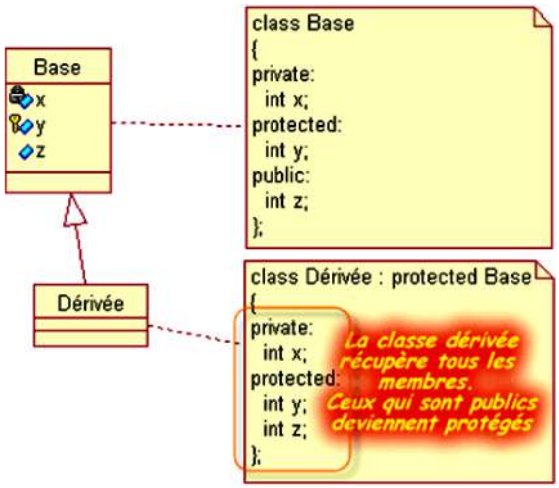
1. Les membres hérités publics et protégés d'une classe de base privée deviennent des membres privés de la classe dérivée.

2. Cette technique permet d'interdire, aux utilisateurs d'une classe dérivée, l'accès aux membres publics de sa classe de base. Cela sous-entend que seules les méthodes de la classe dérivée devront être utilisées, sinon il faudra redéfinir les méthodes de la classe de base (voir plus loin). Pour les dérivations à partir de la classe dérivée, l'accès à la classe de base devient alors totalement inaccessible, les petits enfants n'ont donc pas accès aux membres de leur grands parents.



Dérivation protégée:

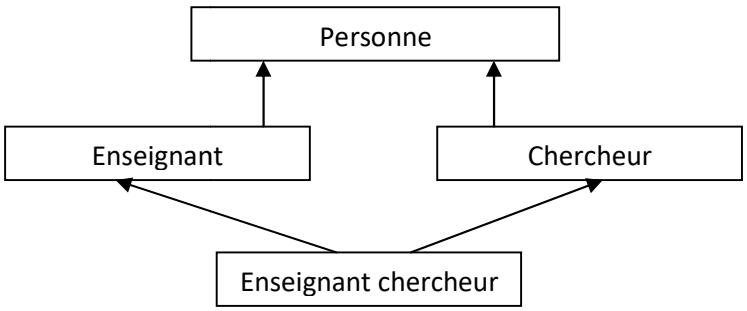
1. Les membres hérités publics et protégés d'une classe de base protégée deviennent des membres protégés de la classe dérivée.
Nous retrouvons le même principe que pour une dérivation privée, la seule différence concerne les enfants éventuels de la classe dérivée, puisque dans ce cas là, les petits enfants peuvent atteindre des membres protégés.



IX.3 Héritage multiple [7]

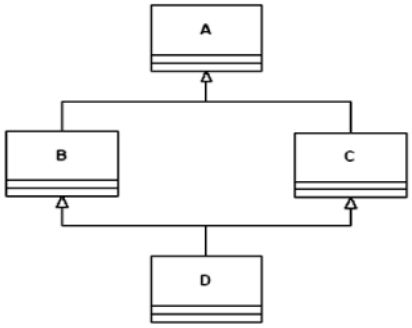
La classe dérivée hérite les attributs et les méthodes à partir de plusieurs classes mères.

Exemple:



Remarque :
L'héritage multiple est possible en C++ mais permet de poser quelques problèmes :

- Si les deux classes mères ont des attributs ou des méthodes de même nom (collision des noms lors de la propagation).
- La classe D hérite deux fois les attributs de A, une fois à travers la classe B et l'autre fois à travers C.



- Java ne supporte pas l'héritage multiple.

Syntaxe :

```
Class  Classe_mere1
{  //…
};
class Classe_mere2
{   //…
};
class Classe_derivee: <mode> Classe_mere1, <mode> Classe_mere2
{   //…
};
```

IX.4 Héritage et constructeur [14]

Si la classe mère contient un constructeur dont l'appel est obligatoire alors le constructeur de la classe dérivée doit obligatoirement appelé celui de la classe mère.
Il faut spécifier le nom et les paramètres du constructeur mère après le prototype du constructeur fils.

Exemple:

```
class Point
{   int x,y;
  public:
    Point(int, int) ;
};

class Pointcolor: public Point
{    int couleur;
  public:
     Pointcolor(int a, int b, int c): Point(a, b)
     { couleur=c; }
};
```

Il est possible de mentionner des arguments par défaut dans *Pointcolor*, par exemple :

```
Pointcolor(int a=0, int b=0, int c=1): Point(a, b)
```

Dans ces conditions, la déclaration :

```
Pointcolor  b(5) ;
```

Entrainera :

- l'appel de *Point* avec les arguments 5 et 0
- l'appel de *Pointcolor* avec les arguments 5, 0 et 1

✍ Hiérarchisation des appels

Soit les classes suivantes:

```
class A                    class B : public A
{    ......                {    ......
  public:                    public:
    A(...) ;                   B(...) ;
    ~A() ;                     ~B() ;
    ....                        ....
};                         };
```

Pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au ***constructeur*** de A, puis le compléter par ce qui est spécifique à B et faire appel au constructeur de B. ce mécanisme est pris en charge par C++ : il n'y aura pas à prévoir dans le constructeur de B l'appel du constructeur de A.

La même application s'applique aux ***destructeur*** : lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A (les destructeurs sont appelés dans l'ordre inverse de l'appel des destructeurs).

Exemple:

```
class Point
{
   int x;
    int y;

    public:
```

```
   Point(int abs=0, int ord=0)
    {  cout <<"++constr. point: "<<abs<<"," <<ord<<endl ;
       x=abs ; y=ord ;
    }
   ~Point()
   { cout <<"-- desctr. point: "<<x<<","<<y<<endl ; }
};

class Pointcolor: public Point
{    int couleur;
  public:
    Pointcolor(int , int , int) ;
    ~Pointcolor ()
    { cout<<"-- destr. pointcol -couleur: "<<couleur<<endl ; }
};

Pointcolor::Pointcolor(int abs=0 ,int ord=0, int c=1):Point(abs, ord)
{   couleur=c;
   cout <<"++constr. pointcol: "<<abs<<"," <<ord<<","<<couleur;
}

void main()
{     Pointcolor a(10,15,3) ;
      Pointcolor b(2,3) ;
      Pointcolor c(12) ;
      Pointcolor *adr;
      adr = new Pointcolor(12,25);
      delete adr ;
}
```

Résultat de l'exécution :

```
++ constr. point: 10,15
++ constr. pointcol: 10,15,3++ constr. point: 2,3
++ constr. pointcol: 2,3,1++ constr. point: 12,0
++ constr. pointcol: 12,0,1++ constr. point: 12,25
++ constr. pointcol: 12,25,1-- destr. pointcol -couleur: 1
-- desctr. point: 12,25
-- destr. pointcol -couleur: 1
-- desctr. point: 12,0
-- destr. pointcol -couleur: 1
-- desctr. point: 2,3
-- destr. pointcol -couleur: 3
-- desctr. point: 10,15

Process returned 0 (0x0)   execution time : 3.318 s
Press any key to continue.
```

Remarque :

Quelque soit les situations, nous disposons toujours des quatre mêmes phases pour la création de l'objet et toujours dans le même ordre.

1. *Allocation mémoire* nécessaire pour contenir tous les attributs que comporte l'objet.
2. *Appel du constructeur de la classe dérivée*. Ce constructeur est appelé mais pas encore exécuté. Appel du constructeur de la classe de base spécifié par la liste d'initialisation en récupérant les bons arguments pour les attributs de la classe de base.

3. *Appel et exécution du constructeur de la classe de base*. A moins que la classe de base soit elle-même une classe dérivée d'une autre classe de base, les instructions qui constituent le corps du constructeur sont exécutées.

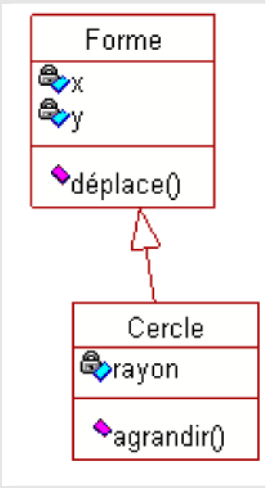
4. *Exécution du constructeur de la classe dérivée*. Puisque la partie générale est bien initialisée, nous pouvons nous occuper de la partie spécifique à la classe dérivée. Les instructions du corps du constructeur sont donc exécutées.

IX.5 Compatibilité entre classe de base et classe dérivée [14]

Nous pouvons toujours dire qu'un cercle est aussi une forme et qu'un élève est aussi une personne. La réciproque n'est pas vraie. Selon le besoin, nous savons établir des conversions implicites qui permettent de passer d'un type vers un autre. Dans le cadre de l'héritage, il sera possible de passer d'une classe dérivée vers une classe de base. Par contre l'inverse ne sera pas possible.

Exemple:

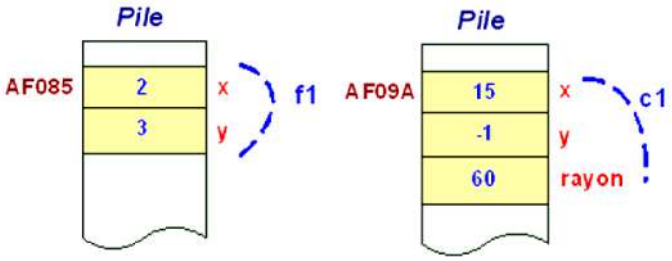
```
class Forme
{   int x;
    int y;
 public:
    Forme(int x, int y)
    { this->x=x ; this->y=y ;}
   void deplacer(int dx, int dy)
    { x+=dx;y+=dy ; }
};

class Cercle: public Forme
{    int rayon;
  public:
     Cercle(int x, int y, int r):Forme(x, y)
      { rayon=r ; }
    void agrandir(int a)
    { rayon+=a; }
};

void main()
{      Forme f1(2,3);   //1
       Cercle c1(15, -1, 50) ; //2
       c1.deplacer(3, 4) ; //3
       c1.agrandir(10) ; //4
       f1=c1 ; //5
       f1.deplacer(5, -8) ; //6
       c1=f1 ; //7
}
```



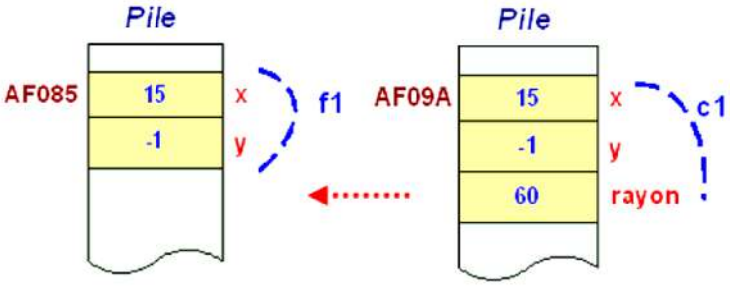
Commentaire:

1. Création de l'objet *f1*.
2. Création de l'objet *c1*.
3. Possibilité de déplacer le cercle *c1* puisque la méthode a été héritée.

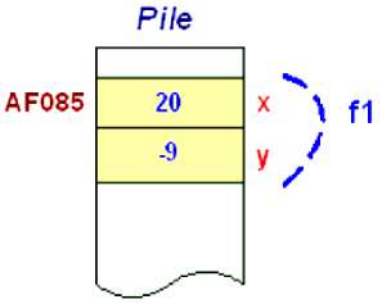
4. Agrandissement du cercle *c1* grâce à la méthode *agrandir* définie par la classe *Cercle*.



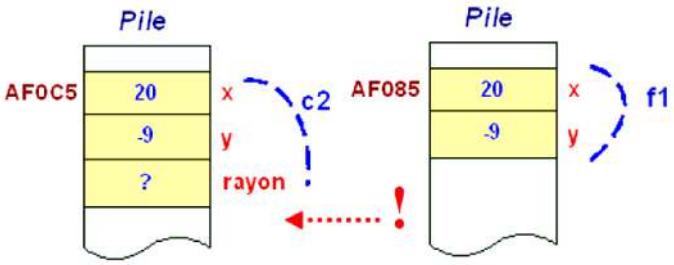
5. A droite et à gauche de l'opérateur d'affectation, les types sont différents. C'est toujours le type qui est à droite qui est transformé vers le type de gauche. Ici, le cercle *c1* est aussi une forme, donc la conversion implicite est lancée. Cette démarche paraît normale puisqu'à l'issue de cette opération, les attributs de l'objet *f1* sont parfaitement définis.



6. Dans ce contexte, il est également possible de changer de position puisque, de toute façon, la méthode associée a été définie dans la classe *Forme*.



7. Tentative d'affectation d'une forme dans un cercle. Nous obtenons également une ***erreur de compilation***. Il s'agit également d'un changement de type. Si ce casting était toléré, cela voudrait dire que nous autoriserions d'avoir des attributs avec des valeurs *aléatoires !*
Effectivement *f1* ne dispose pas de *rayon*, ainsi l'attribut *rayon* de *c2* se retrouverait sans aucune valeur bien précise. Cette démarche n'est pas tolérée par le compilateur et nous le comprenons.



Chapitre X : Polymorphisme

X.1 Etude de Cas [2]

Etudions l'exemple suivant :

```
class A
{   private :
       …..
    public :
       ….
       void f()
       ….
} ;
void A :: f()
{ cout<< "A::f()" <<endl;  }
```

```
class B: public A
{     …
  public:
     void f();
     …
};
void B :: f()
{  cout<< "B::f()" <<endl; }
```

```
void main()
{    A a;
     B b;
     A *p;
     P= &a;  p->f();    // affiche A::f()
     p=&b;   p->f();    // affiche A::f() aussi
}
```

Remarques:

✍ Le choix de la fonction est conditionné par le type de **p** connu au moment de la **compilation**

Puisque p est un A* alors il utilise la méthode A ::f()

Il sera mieux que le deuxième appel écrit B ::f() ? Puisque p contient un B

⇒ Solution : ***Fonction virtuelle***

X.2. Fonction virtuelle [2]

```
class A
{   private :
       …..
    public :
       ….
       virtual void f()
       ….
} ;
```

```
void A ::f()
{ cout<< "A::f()" <<endl; }
```

```
class B: public A
{     …
   public:
      void f();
       …
};
void B::f()
{ cout<< “B::f()” <<endl; }
```

```
void main()
{    A a;
      B b;
      A *p;
      P= &a;  p->f(); // affiche A::f()
      p=&b;   p->f(); // affiche B::f() aussi
}
```

Remarques :

Le choix de la fonction est **maintenant** conditionné par le type exact de l'objet pointé par p connu au moment de **l'exécution** puisque p nous emmène sur un B* alors on utilise B ::f()

X.3 Définition de Polymorphisme

Mécanisme qui consiste à définir des fonctions de même nom dans les classes de bases et les classes dérivées et qui répondent différemment à un même appel. Les classes dérivées héritent tous les membres publics et protégés de la classe mère.

- **En pratique** : Déclarer **virtual** la fonction concernée dans la classe la plus générale de la hiérarchie d'héritage. (Dans l'exemple précédent A)
- Toutes les classes dérivées qui apportent une nouvelle version de f() utiliseront leur fonction à elles.
- Il reste évidement possible d'appeler la fonction f() de A en spécifiant p->A ::f(), mais à ce moment là pourquoi avoir utilisé une fonction virtuelle !?

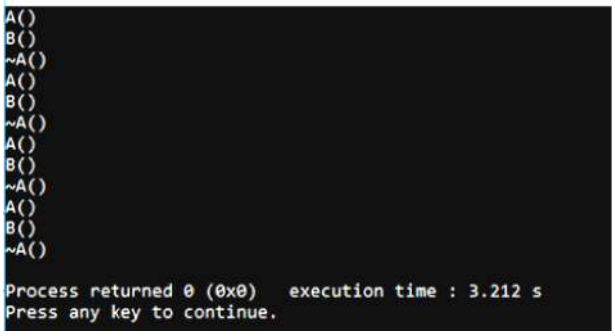
X.4 Destructeur virtuel [2]

Considérons l'exemple suivant :

```
class A
{   protected :
      int *p;

   public :
     A()
      { p=new int[2] ;
        cout<<"A()"<<endl ;
      }
```

```
   ~A()
   { delete [] p ;
     cout<<"~A()"<<endl ;
    }
} ;

class B : public A
{    int *q;
   public :
    B()
    { p=new int[20] ;
      cout<<"B()"<<endl ;
    }
    ~B()
   { delete [] p ;
     cout<<"~B()"<<endl ;
   }
} ;

void main()
{
 for (int I =0; I<4;I++)
   {  A *pa = new B();
      delete pa;
   }
}
```

Affiche ceci :



✍ On doit faire appel au destructeur de B avant celui de A, puisque **pa** référence un ***B.**
Dans ce cas on a un espace mémoire alloué non libéré : (fuite de mémoire)

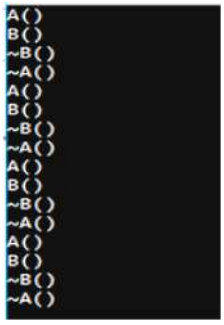
⇒ Solution : **destructeur virtuel**

La classe A devient ainsi :

```
class A
{  protected :
     int *p;

  public :
    A()
     { p=new int[2] ;
       cout<<"A()"<<endl ;
     }
```

```
    virtual ~A()
    { delete [] p ;
      cout<<"~A()"<<endl ;
     }
} ;
```

A l'exécution, on a ceci:



Le bon destructeur est appelé.

X.5 Fonction virtuelle pure – classe abstraite [2]

En POO, nous pouvons définir des classes destinées non pas à instancier des objets, mais simplement à donner naissance à d'autres classes par héritage. On dit qu'on a affaire à des ***«Classes Abstraites».***

En C++, nous pouvons toujours définir de telles classes, en déclarant des fonctions membres virtuelles dont on ne précise pas le contenu dans la classe de base. Seules les classes de base possèdent alors, éventuellement une description du corps de ces fonctions virtuelles. On les appelle des fonctions virtuelles pures.

⇒ Une ***fonction virtuelle pure*** se déclare en remplaçant le corps de la fonction par les symboles = 0.

Syntaxe :

```
class NomClasse
{
   // …
   virtual TypeRetout nomFonction (liste des arguments)=0;
  //…
}
```

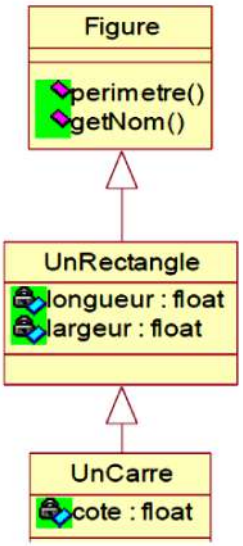
Remarque:

- Une classe comportant au moins une fonction virtuelle pure est considérée comme abstraite et il n'est plus possible de déclarer des objets de son type.

- Une fonction déclarée virtuelle pure dans une classe de base ***doit obligatoirement être redéfinie*** dans une classe dérivée ou déclarée à nouveau virtuelle pure ; dans ce dernier cas, la classe dérivée est aussi abstraite.

Exemple:

```
class Figure
{
  Public:
    virtual float perimetre()=0;
    virtual float surface()=0;
    virtual void dessiner();
}
```



✍ La classe figure est une abstraction. Un programme ne créera pas d'objet *Figure*, mais des objets *Rectangle*, *Cercle*, etc..
On remarque que la fonction *perimetre()* introduite au niveau de la classe *Figure* n'est pas un service rendu aux programmeurs mais une contrainte :

- son rôle n'est pas de dire ce qu'est le périmètre d'une figure, mais ***d'obliger les futures classes dérivées de figure à le dire.***

```
class Figure
{  public:
     virtual float perimetre()=0;
     virtual char * getNom()=0;
};

class UnRectangle:public Figure
{     float longueur;
      float largeur;

  public :
      UnRectangle(float longueur, float largeur)
      {
          this->longueur=longueur;
          this->largeur=largeur;
      }

     float perimetre()
     {  return 2*(longueur+largeur); }

     char* getNom()
     {  return "RECTANGLE"; }
};

class UnCarre:public UnRectangle
{      float cote;

   public:
      UnCarre(float cote):UnRectangle(cote,cote)
      {  this->cote=cote;  }

    float perimetre()
    {  return cote*4;  }

   char* getNom()
   {   return "CARRE";  }
};
```

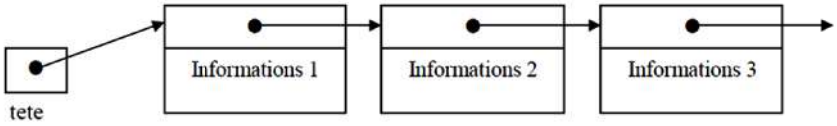
```
void main()
{
   UnRectangle *Rect=new UnRectangle(10,5);
   UnCarre *Carre=new UnCarre(10);
  Figure *T[2];
  T[0]=Rect;
  T[1]=Carre;
  for (int i=0;i<=1;i++)
  {
    cout<<"Le PERIMETRE DU "<<T[i]->getNom()<<" = "<< T[i]->perimetre()<<endl;
  }
}
```

X.6 Exemple d'utilisation de fonctions virtuelles : liste hétérogène [7]

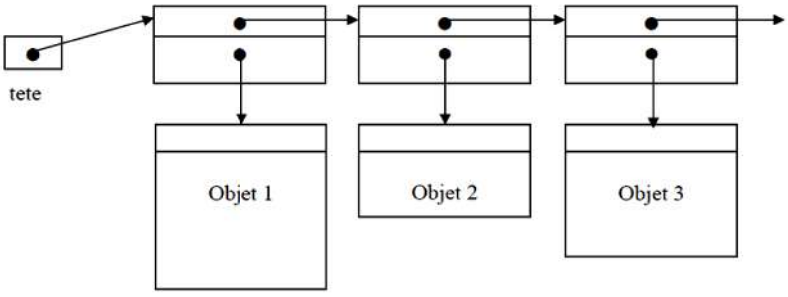
Nous allons créer une classe permettant de gérer une liste chaînée d'objets de types différents et disposant des fonctionnalités suivantes :

- ajout d'un nouvel élément,
- affichage des valeurs de tous les éléments de la liste,
- mécanisme de parcours de la liste.

Rappelons que, dans une liste chaînée, chaque élément comporte un pointeur sur l'élément suivant.



Mais ici l'on souhaite que les différentes informations puissent être de types différents. Aussi cherchons-nous à isoler dans une classe (nommé *liste*) toutes les fonctionnalités de gestion de la liste elle-même sans entrer dans les détails spécifiques aux objets concernés. Nous appliquerons alors ce schéma :



La classe *liste* est composée de :

- un pointeur sur l'élément suivant
- un pointeur sur l'information associée (en fait, ici, un objet).

```
struct element
{   element *suivant ;
    mere *contenu ;
} ;
```

```
class Liste
{    element *tete ;
  public:
     Liste() ;
    ~Liste() ;
    void ajouter(mere *) ;
    void afficher() ;
     .....
};
```

Exemple :

```
class Article
{    long code;
    char nom[100];
    double quantite;
    double prix;

 public:
    virtual void afficher()
    {   cout <<"code="<<code<<endl ;
        cout <<"Nom="<<nom<<endl ;
        cout <<"Quantité="<<quantite<<endl ;
        cout <<"Prix unitaire="<<prix<<endl ;
     }

    virtual void saisir()
    {  cout <<"code="<<endl;              cin>>code ;
       cout <<"Nom="<< endl ;             cin >> nom ;
       cout <<"Quantité="<< endl ;       cin >> quantite ;
       cout <<"Prix unitaire="<< endl ;  cin>> prix ;
     }
};

class Boissons:public Article
{   public:
      double volume;

     void afficher()
      {   cout << "c'est un boissons"<<endl ;
          Article ::afficher() ;
          cout << "volume="<<volume<<endl ;
       }

     void saisir()
     {   cout << "saisie d'un boissons"<<endl ;
         Article ::saisir() ;
         cout << "Volume= "<<endl ;   cin>> volume ;
     }
} ;

class Confiture: public Article
{  public:
    double poids;
```

```
      void afficher()
      {  cout << "c'est un confiture"<<endl ;
         Article ::afficher() ;
         cout << "Poids="<<poids<<endl ;
      }

      void saisir()
      {  cout << "c'est un confiture"<<endl ;
         Article ::saisir() ;
         cout << "Poids= "<<endl ;   cin>> poids ;
      }
} ;

typedef struct element
{   element *suivant ;
    Article *contenu ;
} ;

class Liste
{    element *tete ;
  public :
      Liste()
      { tete=NULL ;}

     ~Liste()
     { delete tete ;}

     void ajouter(Article *a) // au début de la liste
     {   element *nouv= new element ;
         nouv->contenu=a ;
         nouv->suivant=tete ;
         tete=nouv ;
     }

     void afficher()
     {  elemnet *parc=tete ;
        while(parc !=NULL)
        {  parc->contenu->afficher() ;
           parc=parc->suivant ;
        }
     }
} ;

void main()
{
    Confiture c ;
    c.saisir() ;
    Boissons b ;
     b.saisir() ;
     Liste L ;
     L.ajouter(&c ) ;
     L.ajouter(&b ) ;
     L.afficher() ; //affichage de toute la liste
}
```

Chapitre XI : Gestion des exceptions

XI.1 Introduction

Les méthodes traditionnelles de gestion d'erreurs d'exécution consistent à:

- Traiter localement l'erreur : la fonction doit prévoir tous les cas possibles qui peuvent provoquer l'erreur et les traiter localement.
- Retourner un code d'erreur : la fonction qui rencontre une erreur retourne un code et laisse la gestion de l'erreur à la fonction appelante.
- Arrêter l'exécution du programme (*abort*,…)
- Etc. …

C++ introduit la notion d'exception:

- Une exception est l'interruption de l'exécution d'un programme à la suite d'un événement particulier.
- Une fonction qui rencontre un problème *lance une exception*, l'exécution du programme s'arrête et le contrôle est passé à un gestionnaire (*handler*) d'exceptions, qui traitera cette erreur, on dit qu'il *intercepte l'exception*.

Ainsi, le code d'une fonction s'écrit normalement sans tenir compte des cas particuliers puisque ces derniers seront traités par le gestionnaire des exceptions.

La notion d'exception repose donc sur l'indépendance de la détection de l'erreur et de son traitement. Elle permet de séparer le code de gestion de l'erreur et du code où se produit l'erreur ce qui donne une lisibilité et une modularité de traitement d'erreurs.

XI.2 Lancement et récupération d'une exception [8]

- Les exceptions sont levées dans un bloc **try**:

Syntaxe :

```
try
{
   // Un code qui peut lever une ou plusieurs exceptions
   // Les fonctions appelées ici peuvent aussi lever une exception
}
```

- Pour lancer une exception on utilise le mot clé **throw** suivit d'un paramètre caractérisant l'exception :

Syntaxe :

```
throw e; // e est un parametre de n'importe quel type
```

Si le type de e est X, on dit que l'exception levée est de type X.

- Une exception levée est interceptée par le gestionnaire d'exceptions correspondant qui vient juste après le bloc ***try*** et qui est formé par un bloc **catch**.

Syntaxe :

```
catch( type_exception & id_e)
{
   // Code de gestion des exceptions levée par un paramètre de type // type_exception
  // l'argument id_e est facultatif, il représente le paramètre avec // lequel l'exception a été levée
 // le passage de ce paramètre peut être par valeur ou par référence
}
```

Plusieurs gestionnaires peuvent être placés après le bloc try. Ces gestionnaires diffèrent par le type de leur argument. Le type de l'argument d'un gestionnaire précise le type de l'exception à traiter.

Chaque exception levée dans le bloc try, est interceptée par le premier gestionnaire correspondant (dont le type d'argument correspond à celui de l'exception levée) qui vient juste après le bloc try.

C++ permet l'utilisation d'un gestionnaire sans argument, dit aussi *Le gestionnaire universel* qui permet d'intercepter toutes les exceptions. Ce gestionnaire est normalement placé à la fin pour intercepter les gestions qui n'ont pas de bloc catch correspondant:

Syntaxe :

```
catch( ... )
{
   // Code de gestion de l'exception levée par un paramètre de type quelconque
}
```

Exemple [8]***:***
L'exemple suivant montre une utilisation simple des exceptions en C++.

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

//  classe d'exceptions
class erreur
{     const char * nom_erreur;
    public:
       erreur ( const char * s):nom_erreur(s){ }
       const char * raison()
        { return nom_erreur;}
};

//    classe test
class T
{   int _x;
  public:
    T(int x = 0):_x(x)
     { cout << "\n +++ Constructeur\n";}
    ~T()
     { cout << "\n --- Destructeur\n";}
    void Afficher()
     { cout << "\nAffichage : " << _x << endl;}
};
```

```
void fct_leve_exception()          //   fonction qui lève une exception de type 'erreur'
{   cout << "\n----------entree fonction \n";
    throw erreur("Exception de type 'erreur'");
    cout << "\n----------sortie fonction \n";
}
int main()
{  int i;
    cout << "entrer 0 pour lever une exception de type classe\n";
    cout << " 1 pour lever une exception de type entier\n";
    cout << " 2 pour lever une exception de type char\n";
    cout << " 3 pour ne lever aucune exception \n";
    cout << " ou une autre valeur pour lever une exception de type quelconque \n";

    try
    {  T t(4);
      cout << "---> "; cin >> i;
      switch(i)
      {  case 0:
          fct_leve_exception(); // lance une exception de type // 'erreur'
           break;
        case 1:
         {  int j = 1;
           throw j; // lance une exception de type int
         }
         break;
        case 2:
        { char c = ' ';
          throw c; // lance une exception de type char
        }
         break;
        case 3:
         break;
        default:
         {   float r = 0.0;
            throw r; // lance une exception de type float
         }
      }
   }
/******** Gestionnaires des exceptions : aucune instruction ne doit figurer ici ************/
    catch(erreur & e)
    {    cout << "\nException : " << e.raison() << endl;  }

    catch(int & e)
    {   cout << "\nException : type entier " << e << endl;  }

    catch(char & e)
    {  cout << "\nException : type char " << e << endl;   }

    catch(...)
    {  cout << "\nException quelconque" << endl;  }
return 0;
}
```

XI.3 Système d'exceptions [8]

Voici une implémentation de l'opérateur /= permettant de diviser un complexe par un flottant quelconque:

```
complexe& complexe::operator/=(float x)
{  r /= x;
   i /= x;
   return *this;
}
```

Il n'y a ici *aucun traitement d'erreur.* Si on passe 0 à cette fonction, le programme va se planter, mais nous n'avons aucun moyen de récupérer la situation.

✍ Voici une première manière d'introduire un traitement d'erreur:

```
complexe& complexe::operator/=(float x)
{  if ( x == 0 ) throw ( "division par zéro" );
    r /= x;
    i /= x;
    return *this;
}
```

La fonction se contente de "lancer" un **const char***. Celui-ci sera "rattrapé" par une fonction située dans la pile d'appels (c'est-à-dire la fonction appelante, ou la fonction ayant appelé la fonction appelante, etc.) par exemple la fonction main, dont voici une première implémentation:

```
int main()
{
  complexe c(5,6);
  try
  {  float x;
     cout << "Entrez un diviseur: ";
     cin >> x;
     c /= x;
  }
  catch ( const char * c )
  {  cout << c << "\n";  }
return 0;
}
```

✍ La fonction main a "attrapé" l'objet envoyé (ici un const char *) et l'a simplement affiché. La version suivante va plus loin: elle demande à l'utilisateur de rentrer une valeur jusqu'à ce que celle-ci soit différente de 0.

```
int main()
{  complexe c(5,6);
 do
 { try
   {  float x;
      cout << "Entrez un diviseur: ";   cin >> x;
```

```
      c /= x;
      break;
     }
     catch ( const char * msg )
     { cout << msg << " Recommencez\n"; }
} while (true);
return 0;
}
```

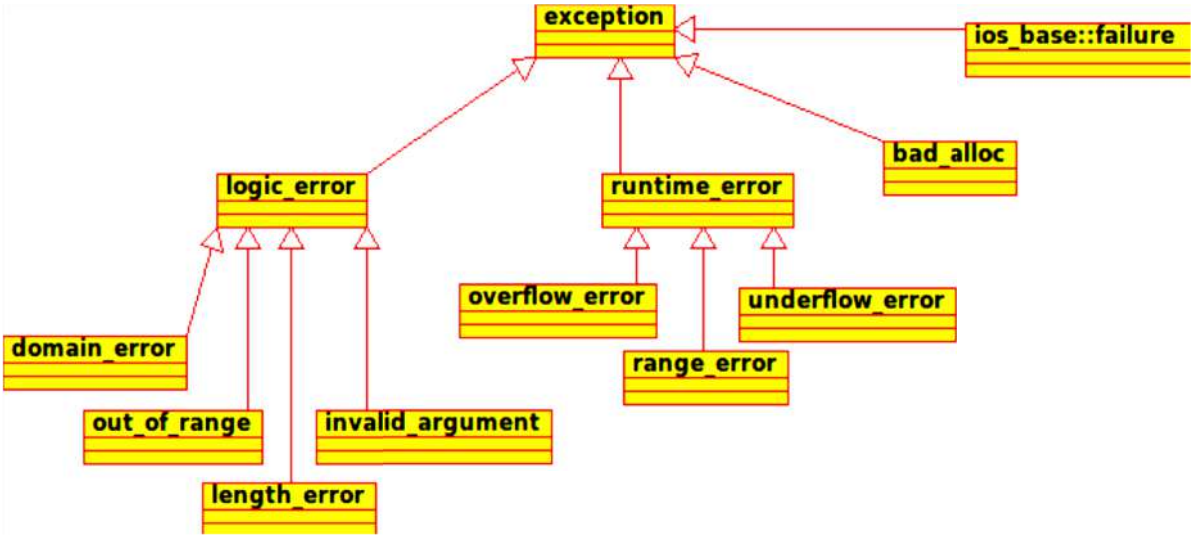
✍ On voit donc ici que si le traitement de l'erreur (dans la fonction main) a changé, la génération de l'erreur, elle, est la même. Le code suivant montre une troisième manière de procéder: tout le traitement d'erreur se fait ici au niveau de la fonction input_et_divise:

```
void input_et_divise(complexe& c)
{ do
  {  try
     {  float x;
        cout << "Entrez un dividende: ";
        cin >> x;
        c /= x;
        break;
      }
      catch ( const char * msg )
      { cout << msg << " Recommencez\n"; }
} while (true);
}

int main()
{  complexe c(5,6);
  input_et_divise(c);
  cout << "Partie reelle : " << c.get_r() << "\n";
  cout << "Partie imaginaire: " << c.get_i() << "\n";
}
```

XI.4 Hiérarchies d'objets exceptions [8]

Plutôt que d'envoyer directement des chaines de caractère, il est beaucoup plus riche d'encapsuler ces messages dans des objets. On peut bien sûr définir ses propres exceptions, mais il est bien plus simple d'utiliser les exceptions déjà définies dans la bibliothèque standard du C++. Si vous préférez définir des objets exceptions, *faites-les dériver de l'une de ces classes* (ne serait-ce que la classe exception).

La figure ci-dessous montre les différentes exceptions définies dans la bibliothèque standard, ainsi que les liens d'héritage qui les relient. La classe de base (exception) possède une méthode abstraite: what(), qui renvoie le message d'erreur encapsulé par l'objet. Lors du throw, on pourra donc générer un message d'erreur suffisamment précis pour que le diagnostic de l'erreur soit aisé.



Nom	Dérive de	Constructeur	Signification
exception		Exception()	Toutes les exceptions dérivent de cette classe
bad_alloc	*Exception*	bad_alloc()	Problème d'allocation mémoire, peut être lancée par l'opérateur new
ios_base::failure	*Exception*	failure(const string&)	Problème d'entrées-sorties, peut être lancé par les fonctions d'entrées-sorties
runtime_error	*Exception*	runtime_error(const string&)	Erreurs difficiles à éviter, en particulier dans des programmes de calcul.
range_error	*runtime_error*	range_error(const string&)	Erreur dans les valeurs retournées lors d'un calcul interne
overflow_error	*Runtime_error*	overflow_error(const string&)	Dépassement de capacité lors d'un calcul (nombre trop grand)
underflow_error	*runtime_error*	underflow_error(const string&)	Dépassement de capacité lors d'un calcul (nombre trop proche de zéro)
logic_error	*Exception*	logic_error(const string&)	Erreur dans la logique interne du programme (devraient être évitables)

domain_error	*logic_error*	domain_error(const string&)	Erreur de domaine (au sens mathématique du terme). Exemple: division par 0
invalid_argument	*logic_error*	invalid_argument(const string&)	Mauvais argument passé à une fonction
length_error	*logic_error*	length_error(const string&)	Vous avez voulu créer un objet trop grand pour le système (par exemple une chaîne plus longue que std::string::max_size()
out_of_range	*logic_error*	out_of_range(const string&)	Par exemple: "index inférieur à 0" pour un tableau

- Il est très simple d'utiliser ces exceptions dans votre programme. L'opérateur précédent peut être réécrit de la manière suivante:

```
complexe& complexe::operator/=(float x)
 {  if ( x == 0 )
    {    domain_error e ("division par zero" );
         throw (e);
    }
    r /= x;
    i /= x;
   return *this;
}
```

- ou encore, de manière plus concise:

```
complexe& complexe::operator/=(float x)
{ if ( x == 0 )
  { throw domain_error( "division par zero" );  }
   r /= x;
   i /= x;
  return *this;
}
```

- Le traitement d'erreur première manière s'écrira cette fois:

```
int main()
{  complexe c(5,6);
   try
   {  float x;
      cout << "Entrez un diviseur: ";   cin >> x;
      c /= x;
   }
  catch ( exception & e )
  {    cout << e.what() << "\n";   }
 return 0;
}
```

- Le traitement d'erreur troisième manière s'écrira comme indiqué ci-dessous. Si une exception de type ***domain_error*** est attrapée par la fonction input_et_divise, elle la traite. Si une autre exception dérivant du type générique exception est émise, elle ne sera pas attrapée par input_et_divise, mais elle sera traitée de manière générique par main.

```
void input_et_divise(complexe& c)
{  do
   { try
     {  float x;
        cout << "Entrez un diviseur: ";
        cin >> x;
        c /= x;
        break;
     }
  catch ( const domain_error& e )
  { cout << e.what() << " Recommencez\n"; }

} while (true);
}
int main()
{  complexe c(5,6);
    try
   {  input_et_divise(c);
      cout << "Partie reelle : " << c.get_r() << "\n";
      cout << "Partie imaginaire: " << c.get_i() << "\n";
   }
catch ( const exception& e )
{ cout << e.what() << "\n"; }
}
```

✍ En fait, plusieurs programmes de capture d'exceptions auraient pu être écrits, suivant la finesse avec laquelle on veut traiter les exceptions:

- On pourrait se contenter de capturer les exceptions de type ***domain_error***
- Dans un traitement plus grossier, on peut capturer les exceptions de type ***logic_error***
- Dans un traitement encore plus grossier, on peut se contenter de capturer les exceptions de type exception.

Il est donc important de passer l'objet exception par const exception &, afin de s'assurer que le bon objet sera au final utilisé (notamment la bonne version de la fonction what()).
Il est plus simple d'utiliser les exceptions prédéfinies, néanmoins il est possible de redéfinir ses propres exceptions.
Dans ce cas, il est important de les définir de manière hiérarchique, et de préférence comme des classes dérivées de la classe exception. Cela permet en effet le traitement hiérarchisé des exceptions, ainsi qu'on vient de le voir.

Références

[1] Programmation Orientée Objet en C++, Fabio Hernandez, CNRS, 2003. http://fr.slideshare.net/airnandez/partie-1-introduction-la-programmation-oriente-objet-en-c

[2] Polycopié de cours POO, TAHAR HAOUET, 2006. http ://www.academiepro.com/uploads/cours/2015_01_03_poo-tahar-haouet.pdf

[3] Programmation Orientée Objet par C++, Zakrani Abdelali, ENSAM, 2016. https://fr.scribd.com/document/295230230/Programmation-Orientee-Objet-par-C-surdefinition-des-operateurs-2015-2016-pdf

[4] Introduction à la programmation en C++, Youssouf El Allioui, 2016.

[4a] https ://fr.slideshare.net/yelallioui/chapitre-03-structures-de-contrôle

[4b] https ://fr.slideshare.net/yelallioui/chapitre-04-les-fonctions

[4c] https ://fr.slideshare.net/yelallioui/chapitre-05-les-tableaux

[5] Programmer efficacement en C++, Meyers S., 2016

[6] La programmation Orientée Objets en C++, A. Al Harraj, 2013/2014. https ://fr.slideshare.net/elharraj/poo-en-c-46530236

[7] L'héritage en C++, Ben Romdan Mourad. https ://fr.scribd.com/document/367452780/cours-Heritage-c

[8] Introduction au C++ et à la programmation objet: Exceptions, Emmanuel Courcelle, 2013 http ://sequence.toulouse.inra.fr/c++/excep.html

[9] Introduction au langage C++: Pointeurs et Références, Christophe Fessard. https ://fr.slideshare.net/fessardnet/6-cours-c-chapitre-pointeurs-et-rfrences

[10] C++ par la pratique, Chappelier J. C. et Seydoux F., 2012

[11] C++ et programmation objet, Mohammed Benjelloun, 2009. https ://moodle.umons.ac.be/pluginfile.php/13797/mod_resource/content/1/info_progobjet_c_2009.pdf

[12] Le langage C++: Initiez-vous à la programmation en C++, Liberty J., Jones B. et Rao S., 2012

[13] JAVA: Support de cours et TD Programmation Orientée Objet, Bouabid M., 2012. http ://filesmbouabid.webnode.fr/200000028-66340672fa/support%20du%20cours_poo.pdf

[14] Programmation JAVA: Héritage simple, Emmanuel Remy.
http ://programmation-java.1sur1.com/c++/pdf/heritagesimple.pdf

[15] Apprendre le C++, Claude Delannoy,2007.

[16] Programmation C++, la classe string, Fresnel.
http ://www.fresnel.fr/perso/stout/langage_c/chap_12_la_classe_string.pdf

[17] Programmation objet en langage C++, Guidet A., 2008

[18] C++ Demystified: A Self-Teaching Guide, Kent J., 2004

[19] Comment programmer en C++, Introduction à la Conception Orientée Objets avec l'UML, Deitel et Deitel, 2003

[20] Programmer en langage C++, Delannoy C., 2000

[21] L'essentiel du C++, Lippman S. B. 1999

[22] Le langage C++, Stroustrup B. 1998

[23] The C++ Programming Language Stroustrup B., 1997

[24] Programming in C++, D'orazio T., 2009

[25] Programmation Orientée Objets, cours/exercices en UML avec C++, Bersini H., 2009