Matière : Conception Orientée Objet

Chapitre 01 : Introduction á l'approche Objet

1- Introduction

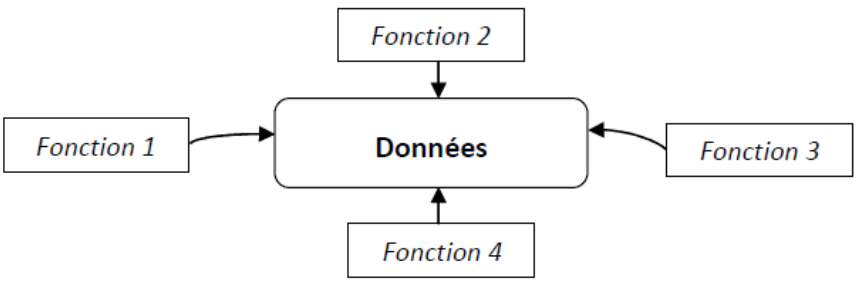
Le paradigme de programmation peut être résumé en une démarche ou une description d'une méthode à mettre en œuvre pour résoudre un problème. Cette démarche est connue sous le nom d'algorithme. En suite, cet algorithme donné sous forme d'instructions est exprimé au moyen d'un langage pour former un programme compréhensible et exécutable par la machine.

Les facteurs de qualité d'un programme peuvent être résumés aux points principaux suivants:

- **l'exactitude** : le programme doit être capable de produire exactement les fonctions qu'on lui demande par les spécifications.
- **la robustesse** : c'est l'aptitude d'un programme à bien réagir lorsqu'on s'écarte des conditions normales d'exécution.
- **l'extensible** : facilité avec laquelle un programme pourra être adapté pour satisfaire une évolution des spécifications.
- **la réutilisabilité** : possibilité d'utiliser certaines parties du code pour résoudre un autre problème.
- **la portabilité** : facilité avec laquelle on peut exploiter un même logiciel sous différents environnements.
- **l'efficacité** : la rapidité d'exécution, la taille mémoire...

1-2- Programmation structurée

La programmation structurée consiste en la résolution d'un problème informatique par l'analyse descendante. Cette technique décompose un problème en sous problème jusqu'à descendre à des actions primitives. On obtient ainsi un programme composé d'un ensemble de sous-programmes appelés procédures (fonctions) qui coopèrent pour la résolution d'un problème et échangent des données qui se trouvent dans des zones séparées à ces fonctions.



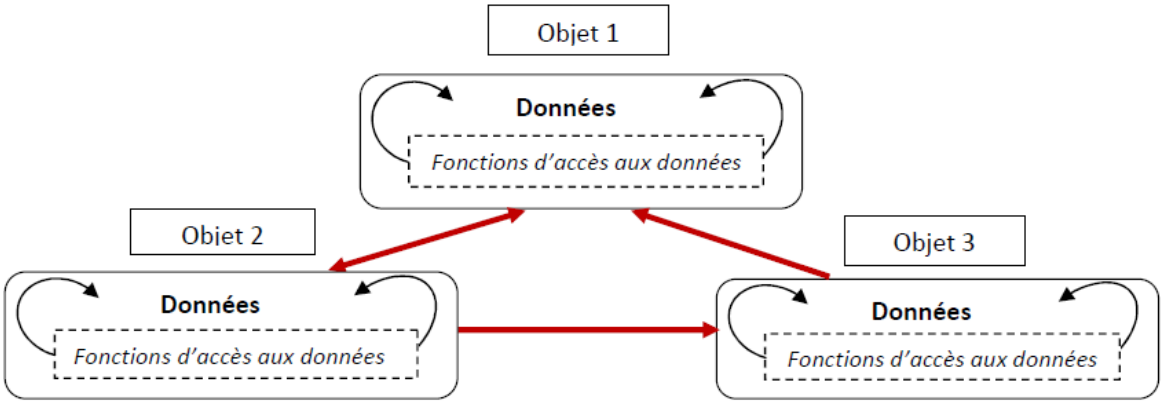
Les inconvénients de la programmation structurée peuvent être résumés comme suit :

- L'évolution d'une application développée suivant ce modèle n'est pas évidente car la moindre modification des structures de données d'un programme conduit à la révision de toutes les procédures manipulant ces données. Ceci est dû à la dissociation qui existe entre ces deux derniers.
- Pour de très grosses applications, le développement peut être très long.

1-3- Programmation orientée objet

Les inconvénients de la programmation procédurales et en particulier la séparation entre données et fonctions ont contribué à développer une nouvelle méthode de programmation appelée Programmation Orientée Objet (POO).

Le concept objet est né du besoin de modéliser dans une même entité les données et les fonctions manipulant ces derniers pour pouvoir simuler des objets du monde réel. Tout système du monde réel est constitué d'un ensemble d'objets qui coopèrent entre eux et avec le monde extérieur. Cette coopération se fait par envoi de messages, à l'aide de divers mécanismes indépendants de l'environnement mis en œuvre.



L'approche objet est donc bien plus qu'une simple technique de programmation, c'est une **manière abstraite** de réfléchir à un problème, en employant des concepts du monde réel plutôt que des concepts purement informatiques.

La **programmation orientée objet** (**POO**) ou programmation par objet, est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de « briques logicielles » appelées objets. Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.

Les éléments qui constituent l'objet définissent à chaque instant l'état de l'objet. Les fonctions quant à elles définissent le comportement de l'objet au cours du temps.

La **POO** se base sur les notions clés suivantes :

- Classe et objets
- Encapsulation
- Abstraction
- Héritage

1-3- 3- Définition d'un objet

L'approche objet consiste à mettre en relation directement les données avec les algorithmes qui les manipulent. Un objet regroupe à la fois des données et des algorithmes de traitement de ces données. Au sein d'un objet, les algorithmes sont généralement appelés par les méthodes et les données (attributs).

OBJET = ATTIBUTS + METHODES

Exemple : un point (un pixel) sur un écran est un objet.

Ses attributs peuvent être :

- Sa position horizontale
- Sa position verticale
- Sa couleur o ....

Ses méthodes :

- positionner en x,y
- Afficher le point

1-3- 4- classe et objet

Une classe est un type abstrait qui encapsulent données et traitement. C'est une sorte de moule qui permet ensuite de créer autant d'instances qu'on veut. Ces instances seront des objets de la classe auxquelles on pourra effectivement envoyer des messages qui activeront les méthodes correspondantes.

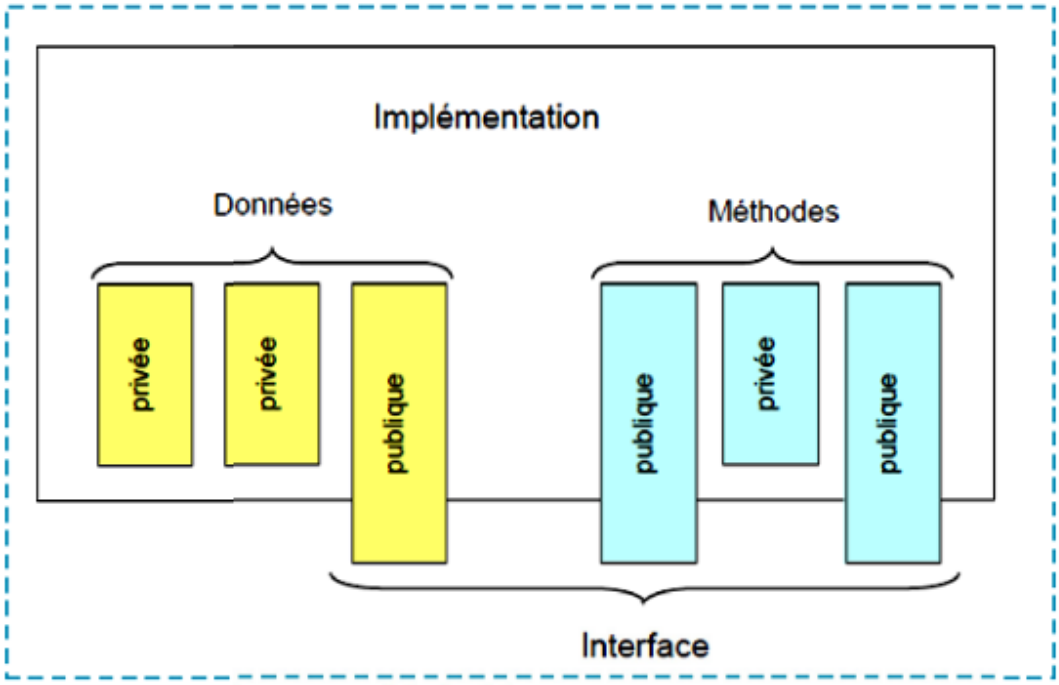
Pour utiliser les objets, il faut d'abord décrire les classes auxquelles ces objets appartiennent. La description d'une classe comporte deux parties :

- une partie déclaration, fiche descriptive des données et fonctions-membres des objets de cette classe, qui servira d'interface avec le monde extérieur.
- une partie implémentation contenant la programmation des fonctions membres.

1-3- 5- Encapsulation

L'encapsulation est le faite qu'un objet renferme ses propres attributs et ses méthodes. Les détails de l'implémentation d'un objet sont masqués aux autres objets. On se trouve ici en face à deux notions : interface et implémentation.

L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.



On précise trois modes d'accès aux attributs d'un objet :

- Le mode **public** avec lequel les attributs seront accessibles directement par l'objet lui même ou par d'autres objets. Il s'agit du niveau le plus bas de protection.
- Le mode **private** avec lequel les attributs de l'objet seront inaccessibles à partir d'autres objets : seules les méthodes de l'objet pourront y accéder. Il s'agit du niveau le plus fort de protection.
- Le mode **protected :** cette technique de protection est étroitement associée à la notion d'héritage (suite du cours).

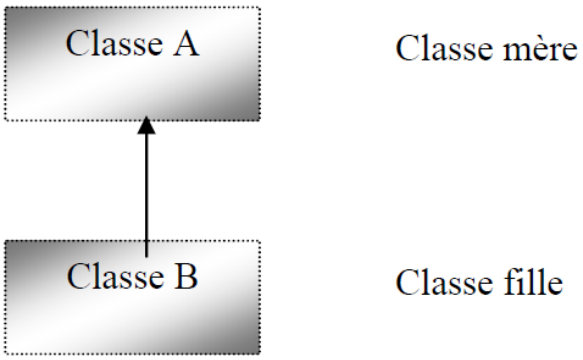
1-3- 6- Abstraction

C'est le faite de se concentrer sur les caractéristiques importantes d'un objet selon le point de vue de l'observateur.

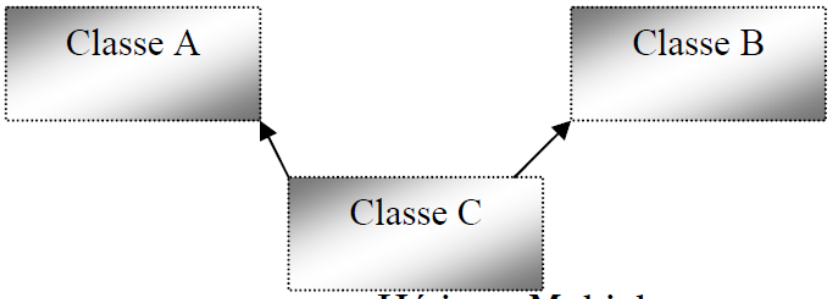
1-3- 7- Héritage

L'héritage consiste à définir différent niveaux d'abstraction permettant ainsi de factoriser certains attributs et/ou méthodes communs à plusieurs classes. Une classe générale définit alors un ensemble d'attributs et/ou méthodes qui sont partagés par d'autres classes, dont on dira qu'elles héritent de (ou spécialisent) cette classe générale.

On parle d'héritage multiple lorsqu'une classe mère possède plusieurs classes filles, et d'héritage simple lorsque la classe fille ne possède qu'une classe mère.



Héritage simple



Héritage Multiple

1. 4. Définition d'un IDE

Le développement de la programmation se fait aujourd'hui grâce à des logiciels appelés **IDE (Integrated Development Environment)** qui peuvent être adaptés à un langage de programmation particulier (Delphi, Lazarus pour le langage Pascal, Visual C++ pour le langage C++) ou à plusieurs langages (Eclipse ou NetBeans pour Java, Php, Javascript).

Un **IDE** comprend en général un éditeur de texte spécialement adapté au langage, un compilateur et/ou un interpréteur, des outils de débogage et surtout un outil de **développement d'interface graphique.** Avant l'existence des IDE, la conception de l'interface graphique d'un programme était extrêmement fastidieuse car le développeur devait lui même préciser dans le programme les positions et dimensions de tous les composants de l'interface. En résumé, les IDE les plus efficaces sont ceux qui permettent:

- la création et la gestion de projets (plusieurs fichiers, plusieurs exécutables, plusieurs librairies, ...),
- l'édition des fichiers avec une reconnaissance de la syntaxe,

- la compilation, l'exécution et le débogue des programmes,
- le *refactoring* de code.

Pour le langage POO C++, Il existe de nombreux IDE on peut citer :

- MS Visual Express
- Dev C++
- wxDevC++, la version de Dev C++ avec un outil de création d'interface graphique (RAD)
- Geany
- NetBeans
- Eclipse, supportant de nombreux langages autre que C/C++
- Code::Blocks
- QtCreator.

1. 5. Les bibliothèques pour interfaces graphiques (GUI)

Il existe de nombreuses bibliothèques permettant de programmer une interface graphique pour un logiciel. La portabilité, la rapidité d'exécution, la rapidité et le coût de développement, la stabilité et la licence du logiciel élaboré dépend essentiellement du choix de la bibliothèque graphique. L'une de ces bibliothèques les plus renommées et utilisées est celle connue sous le nom Qt. En faite, Qt est un **framework multiplateforme** (ensemble énormes de bibliothèques) regroupant un grand nombre d'outils pour développer les programmes plus efficacement. Qt est initialement développé par la société Trolltech, qui fut par la suite rachetée par Nokia. Le développement de Qt a commencé en 1991 et il a été dès le début utilisé par KDE, un des principaux environnements de bureau sous Linux.

Chapitre 02 : Notions de base (Rappels)

II.1. Introduction

Le langage C++ apparu au début des années 80, représente une extension du langage C (né en 1972 par Denis Ritchie), avec des enrichissements tels que les classes, et les modèles (templates). Il représente la version POO du langage C tout en gardant ses notions de base de la programmation.

II. 2. Environnement de développement

Pour pouvoir programmer en C++, il faut y avoir au moins :

- **Un éditeur de texte** pour écrire le code source du programme en C++. Les fichiers source sont des fichiers texte lisibles dont le nom se termine en général par .c, .cpp ou .h.
- **Un compilateur** pour transformer (« compiler ») votre code source en binaire (ou exécutable). Les fichiers exécutables portent en général l'extension .exe sous windows et ne portent pas d'extension sous Linux.
- **Un débugger** (« Débogueur » ou « Débugueur » en français) pour vous aider à traquer les erreurs dans votre programme (un truc qui corrigerait tout seul nos erreurs).

Lorsque ces trois programmes sont regroupés dans un seul programme, on parle d'un **IDE** (ou en français « EDI » pour « Environnement de Développement Intégré »). Il existe plusieurs IDE qui permettent de programmer. Celui utilisé dans ce cours est **Code::Blocks**.

II. 2. 1. Construction de l'exécutable

Le développement d'un programme passe par trois phases successives :

1) écriture et enregistrement des différents fichiers-source,

2) compilation séparée des fichiers .cpp, chacun d'eux donnant un *fichier-objet* portant le même nom, mais avec l'extension .obj,

3) lien des fichiers-objets (assurée par un programme appelé *linker*) pour produire un unique *fichier-exécutable*, portant l'extension .exe ; ce dernier pourra être lancé et exécuté directement depuis le système d'exploitation.

Sur la figure II. 1, on représente les différentes étapes pour construire le fichier exécutable.

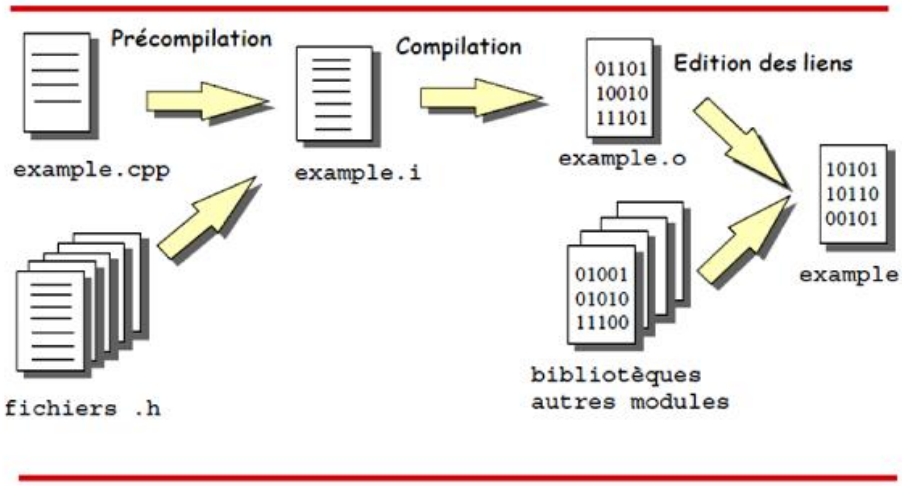


Figure II. 1 : construction du fichier exécutable [1]

II. 2. 1. 1. Fichiers sources

En C++, il y a deux sortes de fichiers-sources pour définir les différentes classes :

- ceux qui contiennent effectivement des instructions ; leur nom possède l'extension .cpp.
- ceux qui ne contiennent que des déclarations ; leur nom possède l'extension .h (signifiant"header" ou *en-tête*).

II. 2. 1. 2. Bibliothèque C++

La bibliothèque standard de C++ regroupe un ensemble de bibliothèques dont la *IO Stream Library* (bibliothèque de flux d'entrée-sortie) et la *STL* ou *Standard Template Library* (bibliothèque standard de modèles) sont les plus importantes. L'utilisation de cette dernière est due à plusieurs raisons telles que la fiabilité, portabilité, efficacité, et compréhensibilité. Pour pouvoir les utiliser il suffit de spécifier avec la directive ***include*** fichier entête (.h) souhaité. En plus, d'autant qu'un programme important peut utiliser de nombreuses bibliothèques, fonctions et objets ou variables et pour éviter les problèmes de conflit de noms, on utilise des espaces de noms (**namespace**).

a- la directive # include : c'est une **directive de préprocesseur**. Son rôle est de « charger » des fonctionnalités du C++ pour que nous puissions effectuer certaines actions (la définition de certains objets, types ou fonctions). Ces extensions qui nous offrent de nouvelles possibilités sont appelées des **bibliothèques**. exp : # include <iostream>

iostream : est pour « *Input Output Stream* » ou bien « Flux d'entrée-sortie », ce que signifie que cette bibliothèque nous permet également de faire plus qu'afficher des messages à l'écran : on pourra aussi récupérer ce que saisit l'utilisateur au clavier.

b- using namespace std : Pour éviter les problèmes de conflit de noms de fonctions, objets,… on utilise des espaces de noms (namespace) :

- un nom est associé à un ensemble de variables, types, fonctions
- leur nom complet est : leur nom d'espace suivi de :: et de leur nom.

remarque:

- Les noms complets des fonctions d'écriture et de lecture cout et cin sont :

std::cout et std::cin.

Pour éviter d'écrire les noms complets, on utilise : using namespace std ;

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

II. 3. Structure générale d'un programme

Tout programme comporte une fonction (et une seule) ou un programme principal, appelée main() et enregistrée main.cpp: c'est par elle que commencera l'exécution. En plus, il y a deux sortes de fichiers-sources pour définir les différentes classes utilisées par la fonction main.

– ceux qui contiennent effectivement des instructions ; leur nom possède l'extension .cpp,

– ceux qui ne contiennent que des déclarations ; leur nom possède l'extension .h (signifiant"header" ou *en-tête*).

Dans cette section, on citera les différentes spécificités liées à la structure générale du C++ et qui ne sont pas obligatoirement relatives à la programmation orientée objet.

II. 3. 1. Fonction main

Elle représente le point d'entrée de tout programme :

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

II. 3. 2. Commentaires

Il existe deux manières pour faire des commentaires :

a- commentaire sur une seule ligne : (spécial C++)

Exemple :

```
//ceci est un commentaire.
int a; // déclaration de a.
```

b- commentaire sur un bloc : délimités par /* (début) et */ (fin).

Exemple :

```
/* ……
   ……
          */
```

II. 4. Variables et déclarations

Une **variable** se caractérise par son **nom**, sa **valeur** et son **type**. En pratique, une variable correspond à une zone de la mémoire centrale. Le C++ est un langage «fortement typé» La compilation permet de détecter des erreurs de typage. Chaque variable d'un programme a un type donné tout au long de son existence. Les types fondamentaux en C++ peuvent représenter une valeur numérique, sur 1, 2 ou 4octets, signé ou non, un nombre à virgule flottante dont l'encodage en mémoire est assez complexe.

La déclaration des variables doit s'effectuer avant leur utilisation n'importe où dans un bloc ou dans une fonction.

Exps :

```
main()
{
      int  a=7,  b;
      b=a;
      float x, y;
      x=2*a+y;
}
```

```
main ( )
{
 int s=0 ;
for (int i=0 ; i<10 ; i++)
s=s+i ;
}
```

Les types dérivés des types fondamentaux sont des types définis par le programmeur, cette dérivation se fait à l'aide d'opérateurs de déclaration. On peut distinguer trois types de dérivation:

- Tableaux.
- Pointeurs.
- Références.

II. 4. 1. Tableaux

Un tableau est un ensemble de types tous identiques. Le nombre de types impliqués dans cet ensemble est appelé **taille** du tableau. L'accès aux composantes s'effectue via un système d'indices entiers numéroté de 0 à taille -1. On parle ici de tableaux statiques.

Syntaxe : type nom_tab [taille] ; // déclaration

ou

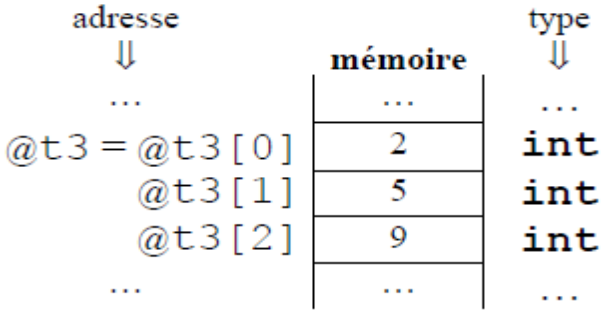
type nom_tab [taille] = {e_1, e_2, …, e_m}; // déclaration et initialisation ($m \leq taille$)

ou

type nom tab [] = {e_1, e_2, …, e_n}; // La taille du tableau peut être omise et
// alors déduite : elle égale le nombre
// d'expressions d'initialisation (n).

exp1 : int t3 [3] = {2, 5, 9}**;**

Du point de vue de l'environnement, les composantes d'un tableau sont toutes rangées dans des cases mémoire adjacentes. L'adresse associée à un tableau est celle de sa première composante.



exp2 : tableau à deux dimensions

int M[2][3] = *{{*1, 2, 3*}*, *{*3, 4, 5*}}*;

ou : int M[2][3] = *{*1, 2, 3, 3, 4, 5*}*;

le tableau M d'entiers à deux indices, le premier indice variant entre 0 et 1, le second entre 0 et 2. On peut voir M comme une matrice d'entiers à 2 lignes et 3 colonnes. Les éléments de M se notent M[i][j].

- déclaration par typedef

Syntaxe générale : typedef <*déclaration*>

Exemple :

const int n = 3; // « n » est une variable de type constante (ne change pas de valeur le long du programme.)

typedef float tab[n]; // définit un type Vecteur (= tableau de trois réels).

- directive # define

Syntaxe générale : # define n 3
float tab[n] ;

II. 4. 1. 2. Tableaux dynamiques

Les tableaux dynamiques 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. Pour pouvoir les utiliser, il faut insérer la ligne : # include < vector>.

Exp [2] :

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
// initialier une matrice Mat(3,4) avec la valeur 5
vector<vector<int> >Mat(3,vector<int>(4,5));
for(int i=0;i<3;i++)
        for(int j=0;j<4;j++)
        {cout<<" "<<Mat[i][j];
        if (j==3)
        cout<<endl;
        }
}
```

Exécution :

5 5 5 5
5 5 5 5
5 5 5 5

- Modification de la taille du tableau : 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.

a. Fonction push_back() :

exp : vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2

tableau.push_back(9); //On ajoute une 4ème case au tableau qui contient la valeur 9

b. Fonction pop_back()

exp : vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2

tableau.pop_back(); // Il y a plus que 2 éléments dans le tableau

remarque:

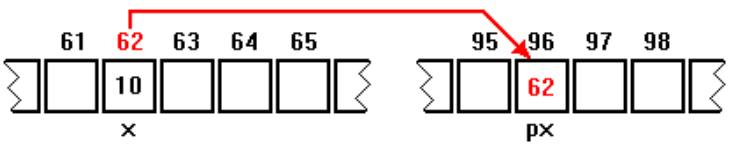
Il existe d'autres opérations de la classe vector qu'on peut les trouvées dans la référence [1].

II. 4. 2. Pointeurs

Un pointeur est une variable qui stocke une adresse vers une zone mémoire (tableau ou variable).

exp : int *px; // déclare la variable px comme pointeur

px = &x; // initialise pointeur px par l'adresse de x.



a. tableaux et pointeurs

Le nom d'un tableau correspond à l'adresse de sa première case. Ce nom alors représente un pointeur contenant l'adresse de la première case du tableau. Soit un tableau **T** d'un type quelconque et *i* un indice pour les composantes de **T** :

T désigne **&T[0]**

T+1 désigne **&T[1]**

T+i désigne **&T[i]**

***(T+i)** désigne **T[i]**

Exp :
```
int tab[3]={1,2,7}; // tableau de 3 int initialisés
int * p=tab ; // identique à p=&tab[0]
p [2]=3 ; // licite : ptrI[2] équivaut ici à tab[2]
p=&tab[1] ; // licite aussi , identique à p=tab+1
p[1]=2 ; // ptrI[1] équivaut maintenant à tab[2]
```

II. 4. 3. Références

Une référence est variable représentant un synonyme - un alias (un nom supplémentaire) – pour désigner l'emplacement mémoire d'une autre variable.

Syntaxe : int & z = x ; // z est une référence a x.

Si on change la valeur de x, la valeur de z est aussi changée et inversement. Idem avec *p.

Exp :
```
int x = 4; // x et z valent 4, *p aussi
int & z = x ; // z est une référence a x.
int *p=&x ;// p est un pointeur sur x.
z = 5; // x et z valent 5, *p aussi
*p = 6; // *p, x et z valent 6
```

II. 5. Fonctions

Toute fonction avant d'être utilisée (appelée) doit être définie ou déclarée puis définie.

Déclaration (prototype): type nom_fct (liste des paramètres) ;

Exp : int max(int ,int) ; // prototype

Définition :
```
type nom_fct (liste des paramètres)
{
    Corps de la fonction
}
```

• type est le type du résultat renvoyé par la fonction (« void » s'il n'y a pas de valeur de retour).

• Liste des paramètres (paramètres formels) de la forme : type1 a_1, . . . , typen a_n,

• Corps de la fonction décrit les instructions à effectuer lors de l'appel de cette fonction.

NB : Lorsqu'une fonction renvoie un résultat, il doit y avoir (au moins) une instruction « return expression ; » avant de fermer l'accolade.

exp : dans le 1er cas la fonction est définie avant l'appel.

Sans prototype :

```
//=========================
// Définition de la fonction max
int max(int a,int b)
{ int res=b;
if (a>b) res = a;
return res;
}
//=========================
//Programme principal
//Appel de la fonction max
int main( )
{ …
int k=34, t=5, m;
m = max(k,2*t+5);
…
}
```

dans le 2ème cas la définition est après l'appel donc on doit la déclarer

Avec prototype :

```
//=========================
// déclaration du prototype
int max(int, int);
//-----------------------
int main( )
{ int k=34, t=5, m;
m = max(k,2*t+5);
//=========================
//=========================
// Définition de la fonction max
int max(int a,int b)
{ int res=b;
if (a>b) res = a;
return res;
}
//=========================
```

II. 5. 1. Arguments de la fonction par défauts

En C++, il est possible de prévoir des valeurs par défauts de certains arguments de la fonction (obligatoirement les derniers).

Exp: float fct(char, int =10, float=1.2)

Ces valeurs par défaut seront utilisées lorsqu'on appelle la fonction :

fct ('a') ;// équivalent à fct('a', 10,1.2)
fct ('a', 12) ; //équivalent à fct ('a', 12, 1.2) ;

II. 5. 2. Passage des paramètres

a. 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.

Exp : //programme [alwafiq maroc]

```
main()
{
void echange(int,int);
int a=2, b=5;
cout<<"Avant appel : "<<a<<" - "<<b<<" \n";
echange(a,b);
cout<<"Après appel : "<<a<<" - "<<b<<" \n";
}
//-------------------------------------------------------
void echange(int m,int n)
{
int z; z=m; m=n; n=z;
}
```

exécution

```
Avant appel :  2 – 5
Après appel :  2 - 5
```

b. passage par adresse (pointeur): 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.

Exp : // programme // exécution

```
main()
{
void echange(int *,int *);
int a=2, b=5;
cout<<"Avant appel : "<<a<<" - "<<b<< " \n ";
echange(&a,&b);
cout<<"Après appel : "<<a<<" - "<<b<< " \n ";
}
//-----------------------------------------------------
void echange(int *x,int *y)
{
int z; z=*x; *x=*y; *y=z;
}
```

```
Avant appel :   2 – 5
Après appel :   5 -  2
```

c. passage par référence: ce mode de passage est identique au passage par adresse mais d'une manière plus simple.

Exp : // programme // exécution

```
main()
{
void  echange(int &,int &);
int a=2, b=5;
cout<<"Avant appel : "<<a<<" - "<<b<<"\n ";
echange(a,b);
cout<<"Apres appel : "<<a<<" - "<<b<<"\n ";
}
//-----------------------------------------------------
void echange (int & x,int & y)
{
int z; z=x; x=y; y=z;
}
```

```
Avant appel :   2 – 5
Après appel :   5 -  2
```

Chapitre III : Classes et Objets

III. 1. Introduction

La POO (Programmation Orientée Objet) est basée sur le principe de briques logiciels appelé Objets. Ainsi, pour faire de la POO, il faut savoir concevoir des classes, c'est à dire définir des modèles d'objets, et créer des objets à partir de ces classes.

III. 1. 1. Notion de classe

Une classe apparait comme un type ou un moule à partir duquel il est possible de créer des objets. Concevoir une classe, revient à définir :

a. **Les données caractéristiques** des objets de la classe. On appelle ces caractéristiques les variables d'instance.

b. **Les actions** que l'on peut effectuer sur les objets de la classe. Ce sont les méthodes qui peuvent s'invoquer sur chacun des objets de la classe.

Une classe -qui définit un type d'objet- possède la structure suivante :

– Son nom (première lettre toujours en majuscule) est celui du type que l'on veut créer, précédé par le mot **class**.

– Elle contient les noms et le type des **données** ou **attributs** (les variables d'instances) définissant les objets de ce type.

– Elle contient les **méthodes** ou les **actions** applicables sur les objets de la classe.

exp :
```
class   Point
 {  double x, y ;// données de la classe Point.
    Void affiche() ;// méthode de la classe Point
 }
```

III. I. 2. Notion d'objet

L'objet est créé à partir d'un modèle appelé classe. Chaque objet créer à partir de cette classe représente une **instance** de cette classe.

Ainsi, chaque objet créé possèdera :

a. Un **état**, c'est à dire des valeurs particulières pour les variables d'instances de la classe auquel il appartient.

2. Des **méthodes** qui vont agir sur son état et décrire son comportement à travers des messages. L'ensemble des messages utilisant cet objet forme ce que l'on appelle l'**interface de l'objet**.

exp : Point a,b ;// a et b représentent des objets de la classes Point.

III. I. 3. Notion de visibilité

Le C++ permet de préciser le type d'accès aux membres d'un objet (attributs et les méthodes). Cette opération s'effectue au sein de la déclaration des classes de ces objets :

Public : les membres publics peuvent être utilisés dans et par n'importe qu'elle partie du programme.

Private : les membres privés d'une classe ne sont accessible que par les objets de cette classe.

remarque : il existe un autre type d'accès protégé (Protected), qu'on le verra plus tard avec la notion d'héritage.

III. I. 4. Notion d'encapsulation

L'encapsulation est l'idée de **protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler**. En respectant ce principe, toutes les variables (attributs) d'une classe seront donc privées.

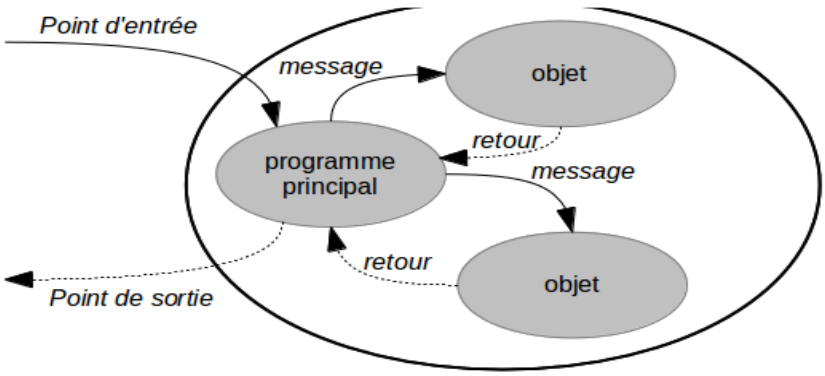
```
exp :    class Point
    {
        private:
           double x, y; // membres privés de la classe Point
        public:
          string nom; // membre public de la classe Point
    };
```

Utilisation :

```
Point point; // instance (ou objet) de la classe Point
point.x = 0.; // erreur d'accès : x est déclarée comme privé
point.y = 0.; // idem
point.nom = "O"; // pas d'erreur car nom est déclaré public
```

III. 1. 5. Notion de message

Les objets interagissent entre eux en **s'échangeant des messages**.



exp :
class Point
{
public:
void afficher(); **//méthode publique de la classe Point _**
};

l'utilisation :

Point point; **// un objet de la classe Point**
// Réception du message "afficher" :
point.afficher(); **// provoque l'appel de la méthode afficher() de la classe Point**

le point « . » utiliser pour l'appel de la fonction pour un objet statique.

III. 2. Règles d'utilisation des fichiers « .cpp » et « .h »

Pour pouvoir réutiliser les classes et les compiler séparément, il faut avoir les trois fichiers suivants :

- **un fichier « .h »** ⟶ qui contient la déclaration de la classe (attributs et méthodes). Il s'appelle l'interface de la classe.

Exp : //fichier Point.h
#ifndef POINT.H
#define POINT.H
#include …..
class Point {
private:
…..
Public:
……
}#endif

- **un fichier « .cpp »** ⟶ qui contient la définition de la classe. Il constitue le corps de la classe. Il contient la définition des différentes méthodes (fonctions) utilisées dans cette classe.

exp : //fichier Point.cpp
#include"Point.h"
#include …..
void Point::initialize(int a, int b)
{x=a;
y=b;}

- **un fichier « main.cpp »** ⟶ c'est le programme principale qui permet l'utilisation des différentes classes crées. Il inclut seulement le fichier « .h » de la classe.

exp : //main.cpp
#include<iostream>

```
#include”Point.h”
…..
main(){
Point p1, p2;
p1.initilise(5, 2);
p2.affiche (5, 4) ;
}
```

II. 1. Définition des fonctions membres de la classe

Toute fonction (méthode) déclarée doit être définit. La définition d'une fonction membre d'une classe est contenue dans le fichier « .cpp » et suit la même syntaxe de la définition de la fonction c, avec un nom préfixé par le nom de la classe et l'opérateur « :: ».

Syntaxe : type nom_classe :: nom_fct(arguments) {……}

exp :
```
void Point ::deplace(int dx, int dy){
    x=x+dx ;
    y=y+dy ;}
```

remarque : « :: » s'appelle l'**opérateur de résolution de portée** et doit précéder la définition de chaque méthode de la classe, pour préciser au compilateur que ce sont des membres de la classe.

II. 2. Accesseurs et Mutateurs

En plus des méthodes créées au sein de la classe, il existe des méthodes publiques spécifiques appelées accesseurs qui permettent d'accéder aux valeurs des données membres (attributs) privées. On pourra même modifier ces valeurs grâce à des fonctions spécifiques appelées mutateurs.

remarque :

-le mot clé « this »désigne un pointeur vers l'instance courante de la classe elle même.

- « this->x » désigne la donnée membre de la classe alors que x désigne le paramètre de la méthode void setX(double x);

- le mutateur double getX() renvoie la valeur de x.

exp :

```
 //fichier Point.h
#ifndef POINT_H
#define POINT_H
class Point
{
public:
void setX(double x);
void setY(double y);
double getX();
double getY();
void saisir();
void afficher();
private:
double x,y;};
#endif
```

```
// fichier Point.cpp
#include "Point.h"
#include <cmath>
#include <iostream>
using namespace std;
void Point::setX(double x)
{
this->x = x;}
void Point::setY(double y)
{
this->y = y;}
double Point::getX()
{
return x;
}
double Point::getY()
{
return y;}
void Point::saisir()
{
cout << "Tapez l'abscisse : "; cin >> x;
cout << "Tapez l'ordonnée : "; cin >> y;
}
void Point::afficher()
{
cout << "L'abscisse vaut " << x << endl;
cout << "L'abscisse vaut " << y << endl;
}
```

III. Création de classe et d'objets :

Les objets constituent les briques de base de la POO. Afin de pouvoir les utiliser et maitriser, on se retrouve face à deux notions :

* le constructeur pour la création et l'initialisation.

*le destructeur.

III. 1. Constructeur d'objet :

Un constructeur d'objet est une méthode particulière d'une classe qui porte toujours le même nom de la classe et utilisé pour créer les objets de la classe [3]. L'objectif de l'utilisation du constructeur se résume en:

- allocation d'emplacement mémoire pour l'objet.
- initialisation des attributs de l'objet avec de bonnes valeurs de départ.

Remarques :

– il peut y avoir plusieurs constructeurs pour une même classe.

– un constructeur n'a jamais de type de retour.(même pas le type void)

- il existe implicitement un constructeur par défaut, son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable. Il est sans paramètre.(Il est fortement recommandé d'écrire soi-même le constructeur par défaut).

exp :

```
Point::Point()
{
x = 0;
y = 0;
}
Point::Point(double x, double y)
{
this->x = x;
this->y = y;
}
```

• Le constructeur par défaut initialise x et y à la valeur 0.

• Pour le deuxième constructeur this->x désigne la donnée membre x de la classe et x désigne le paramètre du constructeur.

• Idem pour this->y.

III. 2. Destructeur d'objet :

Le destructeur est une méthode membre appelée automatiquement lorsqu'une instance (objet) de classe cesse d'exister en mémoire. Son rôle est de libérer toutes les ressources qui ont été acquises lors de la construction (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).

– Un destructeur est une méthode qui porte toujours le même nom que la classe, précédé de "~".

III. 3. Instance de classe ou objet:

Un objet est instance d'une classe (une variable dont le type est une classe). Un objet donc est la réalisation effective d'une classe, alors il occupe de l'espace mémoire. Cet espace peut être alloué de deux manières :

-**statiquement** : comme pour les variables de type de base, on écrira le nom de la classe suivie du nom de l'objet.

exp1: class Point {
……}
⋮
Point a ;

exp2: on peut aussi instancier plusieurs objets de même classe dans un tableau alloué statiquement.

Point groupe [4] ;// groupe est un tableau de 4 objets de la classe Point. Le constructeur par défaut est appelé 4 fois (pour chaque objet Point du tableau) !

```
int i;
cout << "Un tableau de 4 Point : " << endl;
for(i = 0; i < 4; i++)
{
cout << "P" << i << " = "; groupe[i].affiche();
}
cout << endl;
```

- **dynamiquement :** l'allocation dynamique d'un objet est effectué par l'intermédiaire des opérateurs **new** et **delete.** Dans ce cas un pointeur sur la zone mémoire où l'objet a été alloué est retourné.

```
exp :  Point *p3; // pointeur (non initialisé) sur un objet de type Point
p3 = new Point(2,2); // j'alloue dynamiquement un objet de type Point
cout << "p3 = ";
p3->affiche(); // Comme p3 est une adresse, on doit  utiliser l'opérateur -> pour accéder aux membres de cet objet
//p3->setY(0); // je modifie la valeur de l'attribut y de p3
cout << "p3 = ";
(*p3).affiche(); // cette écriture est aussi possible : je pointe l'objet puis j'appelle sa méthode affiche()
delete p3; // ne pas oublier de libérer la mémoire allouée pour cet objet
```

III. 4. Objet passé en paramètre ou retourné

Comme toute variable un objet peut être utilisé comme argument d'appel à une fonction ou méthode, il existe deux types principaux de passage de paramètre.

- **passage par valeur :** il est définit dans le prototype de la fonction avec un nom de classe suivie d'un nom de paramètre. L'objet donné en argument d'appel doit être de même type que le paramètre formel, à l'exécution de la fonction. Le paramètre formel est un autre objet dans le même état que l'objet donné en argument d'appel.

remarque : a et b sont dans le même état ssi les valeurs des attributs coïncident, on dit alors qu'ils sont des objets « clones ».

- **passage par référence :** dans ce cas l'objet est définie dans le prototype de la méthode avec un nom de classe suivi de (&), suivi du nom d'objet.

- **objet retourné** : un objet peut être retourné à la fin d'une fonction ou méthode, si celle-ci définit dans son prototype une classe comme valeur de retour. A ce moment-là, c'est le constructeur par copie qui est appelé et la valeur de retour est un clone de l'objet fabriqué dans la fonction ou méthode et retourné à la fin de celle-ci.

exp : soit le programme qui utilise la classe vecteur et teste si deux vecteur sont orthogonaux ou non.

```
class Vecteur
{
 private :
  int size;
  double* p;
 public :
  Vecteur(){size=0;}//constructeur
par défaut
  Vecteur(int);
  vecteur(const  vecteur& v);
  ~vecteur();
};
 Vecteur :: Vecteur(int taille)
{p=new double[taille=size];
  for (int i=0 ;i<taille ;i++)
    p[i]=0;}
```

```
Vecteur::Vecteur(float abs =0.,float ord = 0.)
{x=abs; y=ord;}
void  Vecteur::affiche()
{cout<<"x = "<< x << " y = "<< y << "\n";}
float det(vecteur a, vecteur b)
{ /
float res;
res = a.x * b.y - a.y * b.x;
return res;
}
```

```
Vecteur :: Vecteur(const  vecteur& v)
{ p=new double[size=v.size];
  for (int i=0 ;i<size; i++)
      p[i]=v[i];}
Vecteur :: ~vecteur()
    {delete[] p;}
void vecteur :: affiche()
{ for (int i=0; i<size; i++)
    cout << p[i] <<"";
    cout << "\n";}
```

```
void main()
{
Vecteur a( 3, 2 );
Vecteur b;
b = a.orthogonal();
if ( prodscalair( a, b ) )
cerr << "Probleme !" << endl;
}
```

remarque :

Une méthode dont le prototype est terminé par const est appelée méthode constante. Dans le corps d'une méthode constante, il est impossible de modifier les données membres de l'objet, de même qu'il est impossible d'appeler des méthodes non-constantes de cet objet. En gros, un objet est en accès lecture seulement dans ses méthodes constantes.

IV. Surcharge des fonctions

La surcharge est la capacité des objets d'une même hiérarchie de classes à répondre différemment à une méthode de même nom, **mais avec des paramètres différents**. En fonction du type et de nombre de paramètres lors de l'appel, la méthode correspondante sera choisie. C'est le compilateur qui va choisir la fonction à appliquer en fonction du type des arguments.

Exp : // Programme

```
void f(int x)
{ cout<<″fonction numéro 1 : ″<<x<<″\n″; }
void f(double x)
{ cout<<″fonction numéro 2 : ″<<x<<″\n″; }
main()
{
int a=2; double b=5.7;
f(a); f(b);
}
```

//Exécution

```
fonction numéro 1 : 2
fonction numéro 2 : 5.7
```

Références bibliographiques

[1] MEDDEBER Lila, ZOUAGUI Tarik. « Programmation Orientée Objets en C++ ». Polycopié de cours

[2] Fatma CHAKER KHARRAT. « La Programmation Orientée Objet C++ ». Cours

[3] Abdelhak-Djamel Seriai. « Introduction à la programmation orientée objet : du C au C++ ». Cours.