Chapitre III

Classes et objets en C++

III.1 Introduction

Dans ce chapitre, nous allons étudier en détail la notion de classe et de l'objet qui représentent le noyau de programmation orientée objet. Plusieurs nouvelles notions seront abordées dans ce chapitre qui permet à l'étudient de glisser de la programmation procédurale à la programmation orientée objet, telles que, la notion de constructeur et de destructeur, les objets constants et dynamiques, les méthodes constantes et les méthodes statiques, les fonctions en .

Une classe est un type de données rassemblées plusieurs attribués (variables simples *int float, tableaux*,…) et des méthodes (fonctions, opérations,..) qui peuvent lire et modifier les attributs. La classe se différenciée de la structure par sa capacité d'autorise ou d'interdire d'accès à ses attribués et ses membres (propriété d'encapsulation des données).

III.2 Syntaxe de la déclaration d'une classe en C++

La syntaxe de déclaration d'une classe en C++ est la suivante :

```
class <Nom de la classe>
{
private :
 /* Les attribués et les méthodes de type 'private' (privé) ne sont pas accessibles que aux membres de la classe et ceux des classes amies (les objets de la classe ne peuvent pas accéder directement aux membres privés de la classe)*/
protected :
    /* Comme presque, le mode 'private', les attribués et les méthodes de type 'protected' (protégé) restent inaccessibles aux objets de la classe, mais ils se comportent comme des membres publiques pour les classes dérivées (héritée) */
 public :
    /* Les attribués et les méthodes de type 'public' (publique) sont accessible de l'intérieur et de l'extérieur de la classe (i.e. ne sont pas accessible seulement aux membres de la classe mais également aux objets de la classe). */
};
```

Les mots clés '*public*', '*private*' et '*protected*' permettent de préciser le type d'accès aux membres de la classe. Les propriétés de chaque type peuvent être résumées comme suit :

- Le mot clé '***private***' (privé) autorise l'accès uniquement aux membres de la classe (i.e. intérieur de la classe) et aux membres des classes amies. **Par contre, les objets de la classe ne peuvent pas accéder directement aux membres privés de la classe**.
- Le mot clé '***protected***' (protégé) presque comme 'private', les attribués et les méthodes de type '*protected*' restent inaccessibles aux objets de la classe, mais ils se comportent comme des membres publiques pour les classes dérivées (héritée, cette notion sera abordée dans le Chapitre IV) */
- Le mot clé '***public***' (publique) autorise l'accès **de l'intérieur et de l'extérieur** aux membres de la classe (i.e. les membres publique sont accessibles aux membres de la classe, aux membres des classes amies et également aux objets de la classe).

Remarques importantes:

1. *Le mots clés 'private' n'est pas obligatoire. Par défaut, tous les attribués et les méthodes d'une classe sont privés (private). Cependant, pour autoriser l'accès de l'extérieur à quelques attribués ou méthodes, il est indispensable de déclarer ces derniers comme 'puclic'.*
2. *Les attribués d'une classe sont généralement déclarés comme privés et les méthodes comme publiques (règle d'encapsulation). Cependant, pour certains cas, il peut être utile de déclarer certaines méthodes comme 'private'.*

Exemple III.1

Dans le programme suivant, la classe '*Etudiant*' à trois attribués privés (*m_nom, m_prenom et m_moy*) et une seule méthode publique ('*affiche_info*') . La classe '*Etudiant*' conserve la règle d'encapsulation.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Etudiant
6   {
7   public:
8   /*La méthode 'affiche_info' est accessible
9    de l'extérieur de la classe */
10      affiche_info()
11      {
12          cout <<"Nom : " << m_nom <<endl;
13          cout <<"Penom : " << m_prenom <<endl;
14          cout <<"Moyenne = " << m_moy <<endl;
15      }
```

```
16  private:
17  /*Les attribués 'm_nom', 'm_prenom'
18  et 'm_moy' ne peuvent etre utilisés
19  que par la méthode 'affiche_info'*/
20       string m_nom="", m_prenom="";
21       float m_moy=0;
22  };
```

III.3 Objets

L'objet est une instance d'une classe, c.à.d. est une variable dont le type est une classe. Les objets sont créés par le(s) constructeur(s) de la classe, comme ils peuvent être supprimés par le destructeur.

III.4 Constructeurs des objets

Les constructeurs sont des méthodes permettent de créer et initialiser les objets.

III.4.1 Constructeur par défaut

En C++, le constructeur par défaut est créé automatiquement par le compilateur, il port toujours le même nom de la classe. Dans l'exemple précédent, le constructeur par défaut de la classe '*Etudiant*' est *Etudiant( )*. Comme il est une fonction, il peut être surchargé comme n'importe quelle autre fonction (voir, chapitre II, surcharge d'une fonction).

Exemple III.2

Le programme suivant montre comment ajouter des constructeurs pour la classe *'Etudiant' développée dans Exemple III.1*.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Etudiant
6   {
7   public:
8      //constructeur par défaut
9       Etudiant(){}
10     //constructeur surchargé
11      Etudiant(string mem1,string mem2, float mem3)
12      {
13          m_nom=mem1;
14          m_prenom=mem2;
15          m_moy=mem3;
16      }
17  /*La méthode 'affiche_info' est accessible
18   de l'extérieur de la classe */
19      affiche_info()
20      {
21          cout <<"Nom : " << m_nom <<endl;
22          cout <<"Penom : " << m_prenom <<endl;
23          cout <<"Moyenne = " << m_moy <<endl;
24      }
```

```
25    private:
26  /*Les attribués 'm_nom', 'm_prenom'
27    et 'm_moy' ne peuvent etre utilisés
28  que par la méthode 'affiche_info'*/
29        string m_nom="", m_prenom="";
30        float m_moy=0;
31  };
```

Ainsi, dans la fonction principale main, il est possile de crée des objets de la classe *Etudiant* en utilisant soit le constructeur par défaut *Etudiant( )* qui ne besoin a aucun paramètre ou le constructeur surchargé *Etudiant(mem1, mem2 , mem3)* qui reçois trois paramètres.

```
32
33
34  int main()
35  {
36      Etudiant Etud01;//objet Etud01
37      Etudiant Etud02(" Zorig","Malik",18);//objet Etud02
38
39      Etud01.affiche_info();
40      Etud02.affiche_info();
41
42      return 0;
43  }
```

L'objet Etud01 est créé par le constructeur par défaut qui ne reçoit aucun paramètre et l'objet Etud02 est créé par le constructeur surchargé qui reçoit trois paramètres (string, string, float). Ainsi l'exécution du programme s'affiche le résultat suivant :

```
Nom :
Penom :
Moyenne = 0
Nom :  Zorig
Penom : Malik
Moyenne = 18
```

III.4.2 Constructeur de copie

Le constructeur de copie est également créé automatiquement par le compilateur, son rôle est de créer un objet dont toutes les valeurs de ces attributs sont copiées d'un autre objet. Il port également le même nom de la classe mais il prend l'objet qui va copier comme un paramètre.

Exemple III.3

Dans le programme suivant, l'objet Etud03 est créé par le constructeur de copie de la classe '*Etudiant*'.

```
34  int main()
35  {
36  Etudiant Etud01;//objet crée par le constructeur par défaut
37  Etudiant Etud02(" Zorig","Malik",18);//objet crée par le
38                                       //constructeur surchargé
39  Etudiant Etud03(Etud02);//objet crée par le constructeur de copie
40  cout <<"ObjetEtud01: "<<endl;
41  Etud01.affiche_info();
42  cout <<"ObjetEtud02: "<<endl;
43  Etud02.affiche_info();
44  cout <<"ObjetEtud03: "<<endl;
45  Etud03.affiche_info();
```

```
46
47          return 0;
48      }
```

Ainsi l'exécution du programme précédent affiche le résultat suivant:

```
ObjetEtud01:
Nom :
Penom :
Moyenne = 0
ObjetEtud02:
Nom :  Zorig
Penom : Malik
Moyenne = 18
ObjetEtud03:
Nom :  Zorig
Penom : Malik
Moyenne = 18
```

III.4.3 Listes d'initialisation des constructeurs

Les constructeurs peuvent êtres initialisées par une écriture plus simple et plus pratique en utilisant la liste d'initialisation. La syntaxe d'utilisation et la suivante :

Nom de la classe (paramètre 1, paramètre 2 ,… paramètre n) **: attribué 1** (**paramètre 1**), **attribué 2** (paramètre 2),……. **attribué n** (**paramètre n**)

{

}

Exemple III.4

Le code suivant montre comment initialiser le constructeur surchargé de la classe '*Etudiant'*

```
//constructeur surchargé par la liste d'initialisation
Etudiant(string mem1,string mem2, float mem3):m_nom(mem1),
m_prenom(mem2),m_moy(mem3)
{
}
```

Remarque :

Vous pouvez comparer les deux implémentations du constructeur surchargé dans l'Exemple III.4 et l'Exemple III.2.

III.5 Destructeur des objets

Le destructeur est une méthode utilisée pour supprimer les objets qui n'utilisent plus dans le programme ce qui permet de libérer des nouveaux espaces mémoires. Le destructeur commence toujours par tilde (~) et porte le même nom de la classe (comme le constructeur), mais il ne peut prendre aucun paramètre, c.à.d. il ne peut pas être surchargé. La syntaxe d'utilisation est la suivante :

~ *Nom de la classe* ( ) ;

Par exemple, le destructeur pour la classe *'Etudiant'* est : ~ *Etudiant ( )* ;

III.6 Objets constants

Comme toutes variables, l'objet (variable de la classe) peut être déclaré comme constant, par conséquent, un objet constant ne peut être modifié. De plus, le compilateur limite l'accès aux méthodes de l'objet constant à l'exception des **constructeurs**, des **destructeurs**, et **des méthodes constantes** (cette dernière sera abordée dans la section suivante). La syntaxe de la déclaration d'un objet constant est la suivante:

const 'nom de la classe' 'nom de l'objet'.

Exemple III.5

```
const Etudiant Etud04 ;// Etud04 est un objet constant
```

III.7 Méthodes constantes

Les méthodes constantes sont des méthodes qui ne sont pas autorisées à modifier les attribués des objets (**pour cela elles sont autorisées d'accéder aux objets constants même si ils appartiennent d'autre classe**). Le compilateur peut identifier ces méthodes par le mot-clé '*const*' qui se trouve à la fin de leur prototype.

Exemple III.6

Dans le programme précédent, la classe '*Etudiant*' a une méthode '*affiche_info*' qui ne sert que l'affichage des valeurs des attribués, alors il est possible de déclarer cette méthode comme une méthode constante comme suit :

```
affiche_info()const
{
    cout <<"Nom : " << m_nom <<endl;
    cout <<"Penom : " << m_prenom <<endl;
    cout <<"Moyedddnne = " << m_moy <<endl;
}
```

Remarque très importante:

Les méthodes constantes peuvent modifier les attribués statiques de leur classe, car ces derniers sont indépendants aux objets de sa classe (voir section suivant).

Exercice III.1 (Exercice récapitulatif III.1) :

Développer une classe nommée 'point' qui caractérise un point (dans le plan bidimensionnel) par les données suivantes :

1. Deux attribués (x, y) de type float qui représentent la position du point ;
2. Constructeur par défaut permet de créer les objets de type 'point' et initialise leur attribués (x, y) par les valeurs (0.5, 0.5).
3. Constructeur surchargé permet de créer les objets de type 'point' et initialise leur attribués (x, y) par des valeurs choisis par l'utilisateur (m, n).

4. Une méthode ‘depalce’ permet de déplacer les objets de type ‘point’ soit par :
   - Un pas fixe (5 sur l’axe x et par 3 sur l’axe y);
   - Un pas choisi par l’utilisateur (k, l).
5. Une méthode ‘affiche’ permet d’afficher la position actuelle des objets de type ‘point’.

Tester la classe point dans la fonction principale ‘main’.

Solution d’Exercice III.1 :

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

class Point
{
public:
   //Constructeur par défaut
    Point():m_x(0.5),m_y(0.5) {}
   //Constructeur surchargé
    Point(float m,float n):m_x(m),m_y(n){}
   //Méthode 'deplace'
     deplace()
     {
      m_x+=5;m_y+=3;
     }
   //Surcharge de la Méthode 'deplace'
     deplace(float k,float l)
     {
      m_x+=k;m_y+=l;
     }
   //Méthode 'affiche'
     affiche(string nom_point) const
     {
cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl;
     }
private:
     float m_x, m_y;
};
```

Teste de classe ‘point’

Dans le programme suivant nous allons tester la classe ‘point’ dans la fonction principale ‘main’

```
int main()
{
Point p1;//objet crée par le constructeur par défaut
Point p2(3,10);//objet crée le constructeur surchargé
p1.affiche("p1");
p2.affiche("p2");
p1.deplace();
p2.deplace(2,5);
p1.affiche("p1");
p2.affiche("p2");
    return 0;
}
```

L’exécution du programme précédent affiche le resultat suivant :

```
"D:\Doc_Abdelmalik\P_O_O\Programmation OrientÚe Objet\Exemple_Prog\Classe_exan
la position du point p1 est :0.5,0.5
la position du point p2 est :3,10
la position du point p1 est :5.5,3.5
la position du point p2 est :5,15
```

III.8 Attribués et méthodes statiques

Nous avons vu précédemment que lorsqu'on déclare plusieurs objets de même classe, chaque objet possède ses propres attribués et méthodes. Dans le programme suivant, nous avons ajouté à la classe 'point' un attribué de type tableau (*rpere*) dont ses cases sont initialisées par la valeur 5.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Point
6  {
7  public:
8
9  Point():m_x(0.5),m_y(0.5){}//Constructeur par défaut
10
11 Point(float m,float n):m_x(m),m_y(n){}//Constructeur surchargé
12 deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
13 deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
14
15
16
17 affiche(string nom_point) const //Méthode 'affiche'
18 {cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
19 private:
20 float m_x, m_y,repere[2]={5,5};
21 };
```

Ainsi, l'instruction :

point p1, p3 ;

conduit à la situation suivante:

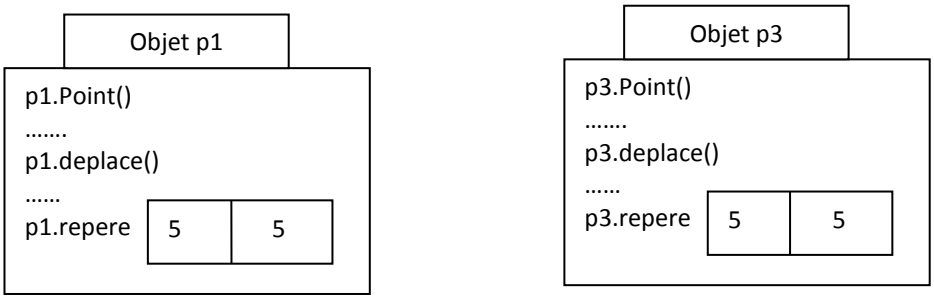

Figure III 1: Attribués des deux objets de même classe.

Par conséquent, le tableau 'repère' est stocké dans chaque objet crée de la classe 'point'. Cependant, il est possible de rendre les objets d'une classe partagent un ou plusieurs attribués/méthodes en utilisant le qualificatif '***static***' (c.à.d. quel que soit le nombre des objets de la classe, il existe une seule copie des donnés (attribués/méthodes) statiques.

Exemple III.7

Dans le programme suivant, les objets de la classe 'point' partagent le même attribué statique 'repere'.

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

class Point
{
public:

Point():m_x(0.5),m_y(0.5){}//Constructeur par défaut

Point(float m,float n):m_x(m),m_y(n){}//Constructeur surchargé
deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'



affiche(string nom_point) const //Méthode 'affiche'
{cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
private:

float m_x, m_y;
static float repere[2];//déclaration d'un attribué statique
};
float Point::repere[2]={5,5};//Initialisation de l'attribué statique
```

Cette situation peut être décrite par le schéma de la figure (III.2)

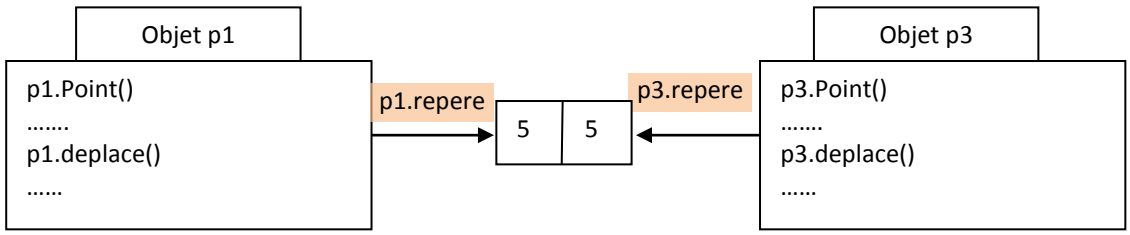

Figure III. 2: Deux objets de même classe partagent un attribué.

Exemple III.8 (méthode statique)

Dans le programme suivant, nous avons ajouté deux méthodes statiques, la première est chargé d'afficher l'attribué statique *'repere'* et la deuxième permet de changer la valeur de l'attribué *'repere'*. Puisque, ces deux méthodes sont de type '*static'* alors tous les objets de la classe 'Point' partagent ces deux fonctions.

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

class Point
{
public:

Point():m_x(0.5),m_y(0.5){}//Constructeur par défaut
Point(float m,float n):m_x(m),m_y(n){}//Constructeur surchargé
deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
affiche(string nom_point) const //Méthode 'affiche'
{cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
```

```
15
16  static affiche_repere() //Méthode statique utilisée pour afficher le repère
17  {
18  cout<<"le  repère de la classe 'Point' est : "<<repere[0]<<" , "<<repere[1]<<endl;
19  }
20
21  static changer_repere(float val_x, float val_y) /*Méthode statique utilisée
22                                                   pour modifier le repère*/
23  {repere[0]=val_x; repere[1]=val_y;}
24
25  private:
26  float m_x, m_y;
27  static float repere[2];//déclaration d'un attribué statique
28  };
29  float Point::repere[2]={5,5};//Initialisation de l'attribué statique
30  int main()
31  {
32  Point p1,p2 (2,2);/* création de deux objets de type 'Point', p1 est créé
33       par le constructeur par défaut et p2 par le constructeur surchargé*/
34  p1.affiche("p1");p2.affiche("p2");//appel à la méthode 'affiche'
35  p1.deplace(); p2.deplace(4,10); //appel auc méthodes:'depalace'et depalce surchargée
36  p1.affiche("p1");p2.affiche("p2");//appel à la méthode 'affiche'
37  Point::affiche_repere();//appel à la méthode statique 'affiche_repere'
38  Point::changer_repere(2.5,2.5);//appel à la méthode statique 'changer_repere'
39  Point::affiche_repere();//appel à la méthode statique 'affiche_repere'
40  }
```

programme test les deux méthode statiques, 'affiche_repere' et 'changer_repere'

L'exécution du programme précédent affiche le résultat suivant :

```
la position du point p1 est :0.5,0.5
la position du point p2 est :2,2
la position du point p1 est :5.5,3.5
la position du point p2 est :6,12
le  repère de la classe 'Point' est : 5 , 5
le  repère de la classe 'Point' est : 2.5 , 2.5
```

Remarques importantes:

1. *L'initialisation des attribués statique doivent être à l'extérieur de la classe sauf si ils sont de type '**static const.** Dans ce cas il suffit de déclaré seulement l'attribué à l'extérieur (sans initialisation).*
2. *Dans le programme précédent, l'accès à l'attribué 'reper' **se fait toujours à travers les méthodes de la classe** puisque cet attribué est privé (principe d'encapsulation).*
3. *Les attribués et les méthodes statiques sont toujours associés à la classe 'Point', pas aux ces objets (indépendant aux objets). Pour cela, lorsqu'on fait un appel à une méthode statique on utilise **seulement le nom de la classe suivi par l'opérateur de résolution de la portée (::) et le nom de l'attribué/la méthode** (voir dans le programme précédent les instructions Point ::affiche_repere et Point ::changer_repere).*

Exercice III.2 (Exercice récapitulatif) :

En considérant la classe 'Point' difinit dans l'exemple précédent, répondre par vraie ou fausse sur les instructions suivantes (justifier vos réponses) :

```
33  Point p1;  //.......?......
34  Point p2 (2,3);//.......?......
35  const Point p3;//.......?......
36  p1.deplace(); //.......?......
37  p2.deplace(); //.......?......
38  p1.deplace(2,4); //.......?......
39  p2.deplace(2,4); //.......?......
40  p3.deplace(2,4); //.......?......
41  p1.affiche(p1); //.......?......
42  p1.affiche("p1"); //.......?......
43  p2.affiche("p2"); //.......?......
44  p3.affiche("p3"); //.......?......
45  Point::affiche_repere();//.......?......
46  Point::changer_repere(4,10);//.......?......
47  p2.affiche_repere();//.......?......
48  p2.affiche_repere();// //.......?.....
49  }
```

Solution d'Exercice III.2 :

```
31  int main()
32  {
33  Point p1;  //vrai, création d'un objet par le constructeur par défaut,
34             // ce dernier n'a pas besoin à aucun paramètre.
35  Point p2 (2,3);//vrai, création d'un objet par le constructeur surchargé,
36             // ce dernier a besoin aux deux paramètres
37  const Point p3;//vrai, création d'un objet constant par le constructeur par défaut
38  p1.deplace(); //vrai, l'objet 'p1' peut utiliser la méthiode déplace
39  p2.deplace(); //vrai, l'objet 'p2' peut utiliser la méthiode déplace
40  p1.deplace(2,4); //vrai, l'objet 'p1' peut utiliser la méthiode déplace surchargé
41                   // (à deux paramètres)
42  p2.deplace(2,4); //vrai, l'objet 'p2' peut utiliser la méthiode déplace
43                   // surchargé (à deux paramètres)
44
45  p3.deplace(2,4); //fausse, p3 est un l'objet constant,alors la méthode deplace
46                   // surchargé ne peut etre utlisée par p3
47
48  p1.affiche(p1); //fausse,la méthode statique 'affiche' a besoin d'un paramètre
49                  //de type string, alors que le parametre p1 dans cette instruction
50                  //est un objet
51
52  p1.affiche("p1"); //Vrai, le parametre "p1" dans cette instruction est de type
53                    //string
55  p2.affiche("p2"); //Vrai le parametre "p2" dans cette instruction est de type
56                    //string
57
58  p3.affiche("p3"); //vrai, la méthode constatnte 'affiche' peut etre utilisée
59                    // par un objet contatant
60
61  Point::affiche_repere();//Vrai la méthode statiqu 'affiche_repere' peut etre
62                          //utilisée par sa classe 'Point'
63
64  Point::changer_repere(4,10);// Vrai la méthode statiqu 'changer_repere' peut etre
65                              //utilisée par sa classe 'Point
66
67  p2.affiche_repere();// Vrai, cette instruction est équivalente à Point::affiche_repere()
68                      // elle s'affiche  4,10
69
```

```
70  p1.affiche_repere();// Vrai, cette instruction est équivalente à Point::affiche_repere()
71                      // elle s'affiche  4,10
```

III.9 Fonctions en lignes

Lorsqu'on fait un appel à une fonction le compilateur réalise quelques taches avant d'exécuter le code de la fonction appelée telles que (stockage des adresses de mémoire, copie des arguments…). Cependant, il est possible de demander de compilateur de remplacer directement l'appel de la fonction par le code de la fonction sans passer par les démarches susmentionnées. Ceci permet d'augmenter l'efficacité du programme ***mais explicitement pour les fonctions courtes*[1].** La syntaxe d'utilisation des fonctions '*inline*' est la suivante :

inline Nom_de_la_fonction (parametres) {//code de la fonction}

Exemple III.9

Dans le programme suivant, nous avons utilisé une fonction '*inline*' pour surcharger le constructeur de la classe 'point'.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Point
6   {
7   public:
8   inline Point(float m=0.5,float n=0.5):m_x(m), m_y(n){}//fonction inline
9   affiche(string nom_point) const /*Méthode 'affiche'*/
10  {cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
11
12  private:
13  float m_x, m_y;
14  static float repere[2];//déclaration d'un attribué statique
15  };
16
17  int main()
18  {
19  Point p1; /*Dans ce cas m_x=0.5 et m_y=0.5 */
20  Point p2 (3);//Cette instruction est possible, m_x=3 et m_y=0.5
21  Point p3 (3,3);/*Dans ce cas m_x=0.5 et m_y=0.5 */
22  p1.affiche("ponit_1");
23  p2.affiche("ponit_2");
24  p3.affiche("ponit_3");
25  }
```

Résultat d'exécution:

```
la position du point ponit_1 est :0.5,0.5
la position du point ponit_2 est :3,0.5
la position du point ponit_3 est :3,3
```

III.10 Pointeur et référence d'un objet

L'utilisation des pointeurs et des références est très utiles avec les classes, car ils permettent d'éviter la recopiée des donnés lors d'une transmission des objets en argument. La syntaxe de la déclaration d'un objet pointeur/référence est similaire à celles des variables ordinaires (int *, int &,..). La syntaxe est la suivante :

[1] L'utilisation des fonctions '*inline*' est recommandée dans le cas où le temps nécessaire pour effectuer l'appel d'une fonction est beaucoup plus long que le temps nécessaire pour exécuter son code d'instructions. C'est pour cette raison leur utilisation est limitée aux fonctions de petite code d'instruction.

Nom_de_la_classe * nom_du_pointeur ; //déclaration d'un objet pointeur

Nom_de_la_classe &nom_de_la_référence (nom_de_la_variable_à_accrocher) /*déclaration d'une référence*/

Exemple III.10

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

class Point
{
public:
//fonction inline
inline Point(float m=0.5,float n=0.5):m_x(m), m_y(n){}
//Méthode 'deplace'
deplace(){m_x+=5;m_y+=3;}
//Surcharge de la Méthode 'deplace'
deplace(float k,float l)    { m_x+=k;m_y+=l;}
/*Méthode 'affiche'*/
affiche(string nom_point) const
{cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }

private:
float m_x, m_y;
static float repere[2];//déclaration d'un attribué statique
};
int main()
{
Point  p1(4,10);//objet
Point  p2(19,10);//objet
Point  *p3=&p1;//objet pointeur pointé à l'objet p1
Point  &p4(p2);//objet référene accrocher à l'objet p2
p1.deplace();
p2.deplace();

(*p3).affiche("p1");// affichage des attribués de l'objet p1
                    // en utilisant le pointeur '*p3'
p4.affiche("p2");//affichage des attribués de l'objet p2
                    // en utilisant la référence '&p4'
}
```

Résultat d'exécution

```
la position du point p1 est :9,13
la position du point p2 est :24,13
```

Exercice III.3 (récapitulant le passage par valeur)

Considérant la classe *'Point'* développée dans *l'Exemple III.8*, en utilisant le passage par valeur développer une méthode '*coincide'* permet de détecter la coïncidence de deux points (points ayant les mêmes coordonnées). Tester ensuite la fonction '*coincide'* dans la fonction main

Solution d'Exercice III.3:

```
#include <iostream>
using namespace std;
class Point
{
public:
Point():m_x(0.5),m_y(0.5){} /*Constructeur par défaut*/
```

```
Point(float m,float n):m_x(m),m_y(n){}/*Constructeur surchargé*/
deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
affiche(string nom_point) const /*Méthode 'affiche'*/
{cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }

 static affiche_repere() /*Méthode statique utilisée pour afficher le repère*/
{
cout<<"le repere de la classe 'Point' est: "<<repere[0]<<" , "<<repere[1]<<endl;
}

static changer_repere(float val_x, float val_y) /*Méthode statique utilisée
                                                  pour modifier le repère*/
{repere[0]=val_x; repere[1]=val_y;}
//Méthode 'coincide' permet de vérifier la coincidence
// de deux point de la classe Point
 void coincide(Point par) const
{
 if  ((par.m_x==m_x)&& (par.m_y==m_y))
 cout<< " OUI"<<endl;
 else
 cout<< " NON"<<endl;
}

private:
float m_x, m_y;
static float repere[2];//déclaration d'un attribué statique
};
float Point::repere[2]={5,5};//Initialisation de l'attribué statique

int main()
{
Point p1,p2(4,10);
p1.coincide (p2); // p1==p2?
p2.coincide (p1);//p2==p1?
p1.deplace(3.5,9.5);
p1.coincide(p2); // p1==p2?
p2.coincide (p1);//p2==p1?
}
```

L'exécution du programme précédent affiche le résultat suivant :

```
"D:\Doc_Abdelmalik\P_O_O\Programmation OrientÚe Objet\Exer
NON
NON
OUI
OUI
```

Exercice III.4 (récapitulant le passage par adresse (pointeur))

Répéter le travail demandé dans *l'Exercice III.3* en utilisant cette fois-ci le passage par adresse (pointeur) pour développer la méthode '*coincide*'. Tester ensuite la fonction 'coincide dans la fonction main.

Solution d'Exercice III.4 :

```
#include <iostream>
#include <iostream>
using namespace std;
class Point
{
public:
Point():m_x(0.5),m_y(0.5){} /*Constructeur par défaut*/
```

```
8   Point(float m,float n):m_x(m),m_y(n){}/*Constructeur surchargé*/
9   deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
10  deplace(float k,float l)   { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
11  affiche(string nom_point) const /*Méthode 'affiche'*/
12  {cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
13
14   static affiche_repere() /*Méthode statique utilisée pour afficher le repère*/
15  {
16  cout<<"le repere de la classe 'Point' est: "<<repere[0]<<" , "<<repere[1]<<endl;
17  }
18
19  static changer_repere(float val_x, float val_y) /*Méthode statique utilisée
20                                                   pour modifier le repère*/
21  {repere[0]=val_x; repere[1]=val_y;}
22  /*Méthode 'coincide' permet de vérifier la coincidence
23  de deux point de la classe Point la fonction 'coincide'
24  utilise le passage par adresse (pointeur)*/
25   void coincide(Point *par) const
26  {
27   /*Les parenthèse dans les instructions  (*par). m_x  et
28   (*par). m_y sont  obligatoires    */
29   if ( (*par).m_x==m_x && (*par).m_y==m_y )
30   //cette écriture st aussi valable (notion de flache):
31    //if ( par->m_x==m_x && par->m_y==m_y )
32
33    cout<< " OUI"<<endl;
34    else
35    cout<< " NON"<<endl;
36  }                                          flèche
37
38  private:
39  float m_x, m_y;
40  static float repere[2];//déclaration d'un attribué statique
41  };
42  float Point::repere[2]={5,5};//Initialisation de l'attribué statique
43
44  int main()
45  {
46  Point p1,p2(4,10);
47  p1.coincide (&p2); // p1==p2?
48  p2.coincide (&p1);//p2==p1?
49  p1.deplace(3.5,9.5);
50  p1.coincide(&p2); // p1==p2?
51  p2.coincide (&p1);//p2==p1?
52  }
```

Le programme précédent affiche le resultat suivant :

```
"D:\Doc_Abdelmalik\P_O_C
 NON
 NON
 OUI
 OUI
```

Remarque

Dans le programme précédent (Exemple III.10), il est possible d'utiliser la notion de flèche (->). Les deux instructions suivantes ***sont équivalentes****:*

```
if ( (*par).m_x==m_x  &&  (*par).m_y==m_y )
if ( par->m_x==m_x  &&  par->m_y==m_y )
```

Exercice III.5 (récapitulant passage par référence)

Refaire le travail demandé dans *l'Exercice III.3* en utilisant le passage par référence pour développer la méthode '*coincide*'. Tester ensuite la fonction '*coincide'* dans la fonction main.

Solution d'Exercice III.5 :

```
#include <iostream>
#include <iostream>
using namespace std;
class Point
{
public:
Point():m_x(0.5),m_y(0.5){} /*Constructeur par défaut*/
Point(float m,float n):m_x(m),m_y(n){}/*Constructeur surchargé*/
deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
affiche(string nom_point) const /*Méthode 'affiche'*/
{cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }

 static affiche_repere() /*Méthode statique utilisée pour afficher le repère*/
{
cout<<"le repere de la classe 'Point' est: "<<repere[0]<<" , "<<repere[1]<<endl;
}

static changer_repere(float val_x, float val_y) /*Méthode statique utilisée
                                                  pour modifier le repère*/
{repere[0]=val_x; repere[1]=val_y;}
/*Méthode 'coincide' permet de vérifier la coincidence
de deux point de la classe Point la fonction 'coincide'
utilise le passage par référence */
 void coincide(Point &par) const
{
 if ( par.m_x==m_x && par.m_y==m_y )

 cout<< " OUI"<<endl;
 else
  cout<< " NON"<<endl;
}

private:
float m_x, m_y;
static float repere[2];//déclaration d'un attribué statique
};
float Point::repere[2]={5,5};//Initialisation de l'attribué statique

int main()
{
Point p1,p2(4,10);
p1.coincide (p2); // p1==p2?
p2.coincide (p1);//p2==p1?
p1.deplace(3.5,9.5);
p1.coincide(p2); // p1==p2?
p2.coincide (p1);//p2==p1?
}
```

Le programme précédent affiche le résultat suivant :

```
NON
NON
OUI
OUI
```

III.11 Membres mutables

La qualificatif 'mutable' est introduite pour relever l'interdiction sur les fonctions constantes à modifier les valeurs des attribués même ces dernier sont d'un objet constant.

Exemple III.11

Dans le programme suivant, la classe '*Circle*' possède deux attribués 'rayon' et 'centre', un constructeur en ligne et deux méthodes constantes, la première permet de calculer la surface et la deuxième permet de modifier l'attribué mutable '*rayon*'.

```
#include <iostream>
using namespace std;
class Circle
{
public:
//Constructeur en ligne
inline Circle(float r=1,float x_centre=0, float y_centre=0):
rayon(r),centre{x_centre,y_centre}{}
/*Méthode en ligne constatne 'surface' permet de calculer la surface
d'un objet type cercle*/
inline float surface()const {return(3.14*rayon*rayon);}

/*Méthode en ligne constatne 'changer_rayon' permet de modifier
le  rayon d'un objet type cercle*/
inline void changer_rayon(float r)const {rayon=r;}/* changer rayon est une
fonction constante mais elle peut modifier l'attribué rayon car ce dernier
est déclaré comme 'mutable'*/
private:
mutable float rayon; // attribué mutable
float centre[2]={0,0};
};
int main()
{
Circle c1;//l'objet c1 de rayon 1 et de centre (0.0)
const Circle c2(4,5,5);//l'objet constant c2 de rayon 4 et de centre (5.5)
cout<<"Surface du c1= "<<c1.surface()<<endl;
cout<<"Surface du c2= "<<c2.surface()<<endl;
c1.changer_rayon(2);/*Comme l'attribué 'rayon' est de type mutable alors,
la fonction 'changer rayon' peut le modifier malgré est une fonction constante*/
c2.changer_rayon(3);/* meme c2 est un objet constant,
'changer_rayone' la fonction 'changer rayon' peut  modifier
son attribué rayon puisque ce dernier est déclaré comme
mutable*/
cout<< "Surface du c1= " <<c1.surface()<<endl;
}
```

L'exécution du programme précédent affiche le résultat suivant :

```
Surface du c1= 3.14
Surface du c2= 50.24
Surface du c1= 12.56
```

III.12 Objets dynamiques

Nous avons étudié au chapitre II, comment réserver/ libérer et manipuler une allocation dynamique de la mémoire pour des variables scalaires (int, float,…) en utilisant les deux opérateurs *'new'* et *'delete'*. Cette notion (allocation dynamique) s'applique de même manière pour réaliser une allocation dynamique de

mémoire pour des objets d'une classe. Considérant par exemple la classe '*Circle'* de l'Exemple *III.11*, si nous déclarons dans '*main'* :

```
Circle * po;
po= new Circle[6];
```

Nous avons créé une allocation dynamique de mémoire pour des objets de la classe *Circle'*. L'accès aux différents membres (attribués/méthodes) se fait soient à l'aide de l'opérateur (*.) ou l'opérateur (->).

Méthode01 : en utilisant l'opérateur (->)

```
int main()
{
Circle * po;
po= new Circle[6];
cout<< "Objet 1= " <<po->surface()<<endl;
*(po+1)=Circle(4,0,0);
cout<< "Objet 2= " << (po+1)->surface()<<endl;
(po+1)->changer_rayon(2);
cout<< "Objet 2= " <<(po+2)->surface()<<endl;
*(po+2)= Circle(5,0,0);
cout<< "Objet 3= " <<(po+2)->surface()<<endl;
}
```

Méthode02 : en utilisant l'opérateur (*)

```
int main()
{
Circle * po;
po= new Circle[6];
cout<< "Objet 1= " <<(*po).surface()<<endl;
*(po+1)= Circle(4,0,0);
cout<< "Objet 2= " <<(*(po+1)).surface()<<endl;
(*(po+1)).changer_rayon(2);
cout<< "Objet 2= " <<(*(po+1)).surface()<<endl;
*(po+2)= Circle(5,0,0);
cout<< "Objet 3= " <<(*(po+2)).surface()<<endl;
}
```

Les deux programmes précédents afficheent le mme résulatat suivant

```
Objet 1= 3.14
Objet 2= 50.24
Objet 2= 12.56
Objet 3= 78.5
```

Chapitre IV

Classes amies, Patrons de fonctions et Surdéfinition des opérateurs

IV.1 Classes amies

Jusqu'à maintenant, nous avons constaté que les membres privés d'une classe (attribués ou méthode) ne sont accessible que aux leur propres méthodes. En outre, le principe d'encapsulation interdit les méthodes d'une classe d'accéder à des donnés privés d'une autre classe. Cependant, dans certains cas nous nous trouvons au besoin d'enfreindre cette règle. Par exemple supposant qu'on a deux classes '*matrice*' et '*vecteur'* et on souhaite développer d'une méthode '*Produit'* permet de calculer le produit vectoriel. *Dans ce cas, la méthode 'Produit' a besoin d'accéder aux donnés des deux classes ('matrice' et 'vecteur') pour calculer le produit.*

La solution est de déclarer la méthode 'Poduit' comme une méthode amie aux deux classe '*matrice'* et '*vecteur'*. La notion d'amitié permet d'autoriser un ou plusieurs méthodes extérieurs d'accès au donnés privés d'une classe.

IV.2 Principaux types d'amitié

En C++, les principales amitiés sont la suivante :

- Méthode indépendante amie d'une classe;
- Méthode d'une classe amie d'une autre classe;
- Méthode amie de plusieurs classes;
- L'ensemble des méthodes d'une classe sont amies d'une autre classe (classe amie d'une autre classe).

IV.2.1 Méthode indépendante amie d'une classe

Dans cette situation, la méthode indépendante (c.à.d. n'appartient à aucune classe) est amie d'une classe. Reprenons, la classe '*Point'* de l'*Exemple III.10. Dans le programme suivant nous avons ajouté une méthode nommée 'permute' (réalise la permutation des coordonnés des deux objets de type 'Point' (m_x, m_y)) amie de la classe 'Point'*.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Point
6  {
7  public:
```

```
8   inline Point(float m=0.5,float n=0.5):m_x(m), m_y(n){}//fonction inline
9   deplace(){m_x+=5;m_y+=3;}//Méthode 'deplace'
10  deplace(float k,float l)    { m_x+=k;m_y+=l;}//Surcharge de la Méthode 'deplace'
11  affiche(string nom_point) const /*Méthode 'affiche'*/
12  {cout<<"la position du point "<<nom_point<<" est :"<<m_x<<","<<m_y<<endl; }
13  //Déclaration de la fonction amie indépendante
14  friend void permutte (Point&, Point&);
15  private:
16  float m_x, m_y;
17  static float repere[2];//déclaration d'un attribué statique
18  };
19
20  //Méthode amie réalise la permuttation des corrdonnées
21  // des deux point*/
22  void permutte(Point &p1, Point &p2)
23  {
24  float par_x(0),par_y(0);
25  par_x= p1.m_x; par_y= p1.m_y;// save p1
26  p1.m_x=p2.m_x;p1.m_y=p2.m_y;// p1=p2
27  p2.m_x=par_x;p2.m_y=par_y;//p2= save
28  }
29
30  int main()
31  {
32  Point point1; // m_x=0.5, m_y=0.5
33  Point point2 (3);// m_x=3, m_y=0.5
34  point1.affiche("point1");//afficher point1
35  point2.affiche("point2");//afficher point2
36  permutte(point1, point2);//appeler la méthode amie 'permutte'
37  point1.affiche("point1");//afficher point1
38  point2.affiche("point2");//afficher point1
39  }
```

L'exécusion du programme précédent affiche les résultats suivants

```
la position du point point1 est :0.5,0.5
la position du point point2 est :3,0.5
la position du point point1 est :3,0.5
la position du point point2 est :0.5,0.5
```

IV.2.2 Classe amie d'une autre classe

Dans cette situation, la classe amie peut utiliser tous les données (attribués +méthodes) de la classe concernées.

Exemple IV.1

Dans le programme suivant, il y a deux classes 'Matrice' et 'Vecteur', en raison que la classe Matrice est amie de la classe 'Vecteur', la classe 'Matrice', peut utiliser toutes les attribués et les méthodes de la classe 'Vecteur'. Par exemple, la méthode 'Produit_M_V' appartient à la classe 'Matrice', son rôle est de calculer le produit d'un objet de la classe 'Matrice' avec celui d'une classe 'Vecteur'. Ceci est possible et ne viole pas le principe de la programmation orientée objet (l'encapsulation) puisque la classe 'Matrice' est déclarée comme une classe amie à la classe 'Vecteur' (voir ligne 38 dans le programme). Par contre, la classe 'Vecteur' ne peut pas utiliser les attribués de la méthode 'Matrice' puisque 'Vecteur' n'est pas amie à la classe 'Matrice'.

```
#include <iostream>
#include <string>
using namespace std;
class Vecteur
{
public:
Vecteur (int L) //Constructeur
{
 if (L<=0 || L>=10)//vérification de la taille
 {
 cout<<"Taille invalide (Min=1, Max= 10)"<<endl;
 v_L=0;
 }
 else
{v_L=L;}
}

//Méthode permet de saisir les composants d'un vecteur:
void Saisie_V()
{
cout << "Saisie du vecteur" <<endl;
for(int i=0;i<v_L; i++)
  {
    cout <<" Vecteur["<<i<<"]= ? " ;
    cin >> V[i];
  }
}
//Méthode permet d'afficher un vecteur:
void Affiche_V()
{
 cout<<"Vecteur = [ ";
 for(int i=0;i<v_L; i++)
  {
    cout << V[i]<<' ';
  }
  cout << "]"<<endl;
}
friend class Matrice;
private:
int v_L=0 ;
int V[10];
};

class Matrice
{
public:
Matrice (int L) //Constructeur
{
if (L<=0 || L>=10)//vérification de la taille de la matrice
{cout<<"Taille invalide,(Ligne X colonne ) doivent etre 1 et 10"<<endl;}
else
{ m_L=L; }
}
//Méthode permet de saisir les composants d'une matrice:
void Saisie_M()
{
cout << "saisie d'une matrice" <<endl;
for(int i=0;i<m_L; i++)
```

```
59    {
60     for(int j=0;j<m_L;j++)
61      {
62      cout <<" Matrice ["<<i<<"]["<<j<<"]= ? " ;
63      cin >>  M[i][j];
64      }
65    }
66  }
67  //Méthode permet d'afficher une matrice:
68  void Affiche_M()
69  {
70   cout<<"Matrice = "<<endl;
71   for(int i=0;i<m_L; i++)
72    {
73     for(int j=0;j<m_L;j++)
74      {
75      cout << M[i][j]<<"\t";
76      }
77   cout<< endl;// ou '\n'
78    }
79  }
```

```
80  //Méthode amie calcule et affiche le produit: Matrice X Vecteur
81    void Produit_M_V (Vecteur Vect )
82    {
83  if ( m_L !=  (Vect.v_L ))// Vérificatioon (v_L est égal à m_L ?)
84   {cout <<" la matrice et le vecteur doivent avoir le meme rang"<<endl;}
85  else
86   {
87      Vecteur VerPr (m_L);//
88     for (int i=0; i< m_L ; i++)// calcul du produit VerPr= M*Vect
89     {
90      VerPr.V[i]=0;
91         for (int j=0; j<m_L; j++)
92          {VerPr.V[i]+= (M[i][j] )*( Vect.V[j] );}
93     }
94     VerPr.Affiche_V() ;
95   }
96  }
97
98  private:
99  int m_L=0;
100 int M[10][10];
```

Exemple d'éxécusion 01 : Le programme suiavnt test les deux classes 'Matrice' et 'Vecteur'

```
105  int main()
106 {
107
108  Vecteur V1(3);
109  V1.Saisie_V();
110  V1.Affiche_V();
111  Matrice M1(3);
112  M1.Saisie_M();
113  M1.Affiche_M();
114  M1.Produit_M_V( V1);
115
116  return 0;
117 }
```

Résultat d'exécution

```
Saisie du vecteur
 Vecteur[0]= ? 3
 Vecteur[1]= ? 3
 Vecteur[2]= ? 3
Vecteur = [ 3 3 3 ]
saisie d'une matrice
 Matrice [0][0]= ? 3
 Matrice [0][1]= ? 3
 Matrice [0][2]= ? 3
 Matrice [1][0]= ? 3
 Matrice [1][1]= ? 3
 Matrice [1][2]= ? 3
 Matrice [2][0]= ? 3
 Matrice [2][1]= ? 3
 Matrice [2][2]= ? 3
Matrice =
3       3       3
3       3       3
3       3       3
Vecteur = [ 27 27 27 ]
```

Exemple d'éxécusion 02

```
105   int main()
106   {
107
108   Vecteur V1(3);
109   V1.Saisie_V();
110   V1.Affiche_V();
111   Matrice M1(2);
112   M1.Saisie_M();
113   M1.Affiche_M();
114   M1.Produit_M_V( V1);
115
116   return 0;
117   }
```

Résultat d'exécution

```
Saisie du vecteur
 Vecteur[0]= ? 4
 Vecteur[1]= ? 4
 Vecteur[2]= ? 4
Vecteur = [ 4 4 4 ]
saisie d'une matrice
 Matrice [0][0]= ? 2
 Matrice [0][1]= ? 2
 Matrice [1][0]= ? 2
 Matrice [1][1]= ? 2
Matrice =
2       2
2       2
 la matrice et le vecteur doivent avoir le meme rang
```

IV.2.3 Méthode amie de plusieurs classes

Il est possible que la même méthode (indépendante ou non) soit en amitié avec plusieurs classes en ajoutant la déclaration d'amitié (freind…) dans toutes les classes qui deviendront amis à cette méthode.

Exemple IV.2

Dans l'exemple précédent *Exemple IV.1*, la méthode *'Produit_M_V'* qui calcul le produit d'une matrice avec un vecteur est appartient à la classe 'Matrice'. Dans le programme suivant nous avons rendre cette fonction

indépendante et amie aux deux classe ‘Matrice’ et ‘Vecteur’. Par conséquent, cette dernière peut accéder à tous les attribués des deux classes Matrice’ et ‘Vecteur’.

```
#include <iostream>
#include <string>
using namespace std;
class Matrice;/*la déclaration de la classe 'Matrice' est
 nécessaire puisque elle est citée avant sa définition */

class Vecteur
{
public:
Vecteur (int L) //Constructeur
{
 if (L<=0 || L>=10)//vérification de la taille
 {
 cout<<"Taille invalide (Min=1, Max= 10)"<<endl;
 v_L=0;
 }
 else
{v_L=L;}
}

//Méthode permet de saisir les composants d'un vecteur:
void Saisie_V()
{
cout << "Saisie du vecteur" <<endl;
for(int i=0;i<v_L; i++)
  {
    cout <<" Vecteur["<<i<<"]= ? " ;
    cin >> V[i];
  }
}
//Méthode permet d'afficher un vecteur:
void Affiche_V()
{
 cout<<"Vecteur = [ ";
 for(int i=0;i<v_L; i++)
  {
    cout << V[i]<<' ';
  }
 cout << "]"<<endl;
}

friend void Produit_M_V (Vecteur, Matrice );
private:
int v_L=0 ;
int V[10];
};

class Matrice
{
public:
Matrice (int L) //Constructeur
{
if (L<=0 || L>=10)//vérification de la taille de la matrice
{cout<<"Taille invalide,(Ligne X colonne ) doivent etre 1 et 10"<<endl;}
else
{ m_L=L; }
}
//Méthode permet de saisir les composants d'une matrice:
```

```
59  void Saisie_M()
60  {
61  cout << "saisie d'une matrice" <<endl;
62  for(int i=0;i<m_L; i++)
63    {
64     for(int j=0;j<m_L;j++)
65      {
66      cout <<" Matrice ["<<i<<"]["<<j<<"]= ? " ;
67      cin >>  M[i][j];
68      }
69    }
70  }
71  //Méthode permet d'afficher une matrice:
72  void Affiche_M()
73  {
74   cout<<"Matrice = "<<endl;
75   for(int i=0;i<m_L; i++)
76     {
77      for(int j=0;j<m_L;j++)
78       {
79       cout << M[i][j]<<"\t";
80       }
81    cout<< endl;// ou '\n'
82     }
83  }
84  private:
85  friend void Produit_M_V (Vecteur, Matrice );
86  int m_L=0;
87  int M[10][10];
88
89  };
90  //Méthode amie calcule et affiche le produit: Matrice X Vecteur
91    void Produit_M_V (Vecteur Vec, Matrice Mat )
92     {
93  if ( Mat.m_L !=  (Vec.v_L ))// Vérificatioon (v_L est égal à m_L ?)
94   {cout <<" la matrice et le vecteur doivent avoir le meme rang"<<endl;}
95  else
96   {
97      Vecteur VerPr (Mat.m_L);
98     for (int i=0; i< Mat.m_L ; i++)// calcul du produit VerPr= M*Vect
99     {
100     VerPr.V[i]=0;
101        for (int j=0; j<Mat.m_L; j++)
102         {VerPr.V[i]+= (Mat.M[i][j] )*( Vec.V[j] );}
103    }
104    VerPr.Affiche_V() ;
105  }
```

Test du programme

```
106
107  int main()
108 {
109  Vecteur V1(3);
110  V1.Saisie_V();
111  V1.Affiche_V();
112  Matrice M1(3);
113  M1.Saisie_M();
114  M1.Affiche_M();
115  Produit_M_V( V1, M1);
116
117  return 0;
118  }
```

Résultat d'exécution

```
Saisie du vecteur
 Vecteur[0]= ? 5
 Vecteur[1]= ? 5
 Vecteur[2]= ? 5
Vecteur = [ 5 5 5 ]
saisie d'une matrice
 Matrice [0][0]= ? 1
 Matrice [0][1]= ? 1
 Matrice [0][2]= ? 1
 Matrice [1][0]= ? 1
 Matrice [1][1]= ? 1
 Matrice [1][2]= ? 1
 Matrice [2][0]= ? 1
 Matrice [2][1]= ? 1
 Matrice [2][2]= ? 1
Matrice =
1       1       1
1       1       1
1       1       1
Vecteur = [ 15 15 15 ]
```

IV.2.4 Classe amie d'une autre classe

Si la méthode amie appartient d'une autre classe (ni pas indépendante), il est nécessaire de préciser sa classe dans la déclaration d'amitié en utilisant l'opérateur de la résolution de la portée (::). Dans l'exemple suivant, nous allons

IV.3 Surdéfinition d'opérateurs pour des classes

Nous avons vu précédemment comme sudéfinit un opérateur pour une structure, voir chapitre II, voir paragraphe xx. Dans cette section, nous allons voir que cette notion peut être appliquée de manière presque similaire avec les classes.

IV.3.1 Surdéfinition d'un opérateur ?

Elle consiste à définir au compilateur comment réaliser des opérations spécifiques sur des types de donné complexes (par exemple structures ou classes) telles que (classe + classe), (classe - classe), (classe && classe)…etc. Autrement dit, elle consiste à développer des méthode permettent de réaliser des opérations (arithmétique (+, - , /, %, =+, ++ ,...) logique (&&, | |, !...) , de comparaison (>, ==, <…). …etc) sur des types de donné complexe (structures, objets,..). Dans cette section nous allons présenter la syntaxe de la surdéfinition d'un opérateur avec quelques exemples.

IV.3.2 Syntaxe de la surdéfinition d'un opérateur

La syntaxe de la surdéfinition d'un opérateur en C++ est la suivante :

Type_de_retour **operator** **symbole** (**parametre 1**, **parametre 2**, … **parametre** n)

Où :

- type de retour peut être un objet d'un classe, un pointeur d'un objet , une référence d'un objet, void….etc.
- Symbole peut être +,-, %, >, ++ ….etc.
- Paramètres peuvent être des donnés simple (int, float) ou complexe (fonction, structure, objet d'une classe….).

Exemple IV.3

Reprennent la classe 'Vecteur' dans le programme de l'Exemple III.14. Dans le programme suivant nous allons surdéfinit un opérateur de l'addition (+) pour la classe 'Vecteur', ainsi la somme des objets de classe 'Vecteur' de même taille (vecteur 1 +vecteur 2+…+vecteur n) sera possible après la surdéfinition de cet opérateur.

```
#include <iostream>
#include <string>
using namespace std;
class Matrice;/*la déclaration de la classe 'Matrice' est
 nécessaire puisque elle est citée avant sa définition */

class Vecteur
{
public:
Vecteur (int L) //Constructeur
{
 if (L<=0 || L>=10)//vérification de la taille
 {
 cout<<"Taille invalide (Min=1, Max= 10)"<<endl;
 v_L=0;
 }
 else
{v_L=L;}
}

//Méthode permet de saisir les composants d'un vecteur:
void Saisie_V()
{
cout << "Saisie du vecteur" <<endl;
for(int i=0;i<v_L; i++)
  {
    cout <<" Vecteur["<<i<<"]= ? " ;
    cin >> V[i];
  }
}
//Méthode permet d'afficher un vecteur:
void Affiche_V()
{
 cout<<"Vecteur = [ ";
 for(int i=0;i<v_L; i++)
  {
    cout << V[i]<<' ';
  }
  cout << "]"<<endl;
}

friend void Produit_M_V (Vecteur, Matrice );
// l'opérateur + est défini comme méthode amie
friend Vecteur operator+(Vecteur, Vecteur);
private:
int v_L=0 ;
int V[10];
};
/*surchage d'un opérateur+ pour la calsse 'Vecteur'*/
Vecteur operator+(Vecteur vec1, Vecteur vec2)
{
   if  (vec1.v_L !=vec2.v_L)
```

```
53      {cout << "les vecteurs doivent avoir la meme taille"<<endl;}
54      else
55      {
56          Vecteur vec3(vec1.v_L);
57          for(int i=0;i<(vec1.v_L); i++)
58           {
59               vec3.V[i]=vec1.V[i]+vec2.V[i];
60           }
61           return vec3;
62      }
63
64  }
```

Exemple d'exécution

Après la surdéfinition de l'opérateur '+' pour la classe 'Vecteur' on peut le tester dans la fonction prinicipale main comme comme dans l'example suivant :

```
int main()
{
Vecteur V1(3);
V1.Saisie_V();
Vecteur V2(3);
V2.Saisie_V();
Vecteur V3(3);
V3= V1+V2;// utilisation de l'opérateur surdéfini
V3.Affiche_V();
V3= V1+V2+V1+V2;// utilisation de l'opérateur surdéfini
V3.Affiche_V();
return 0;
}
```

Résultat d'éxécution :

```
Saisie du vecteur
 Vecteur[0]= ? 1
 Vecteur[1]= ? 1
 Vecteur[2]= ? 1
Saisie du vecteur
 Vecteur[0]= ? 2
 Vecteur[1]= ? 2
 Vecteur[2]= ? 2
Vecteur = [ 3 3 3 ]
Vecteur = [ 6 6 6 ]
```

Remarques importantes

1- *Dans le programme précédent (Exemple III.10), la surdéfintion de l'opérateur + pour la classe 'Vecteur' est implémentée comme une classe amie. Néanmoins, il est possible de l'implémentée également comme une simple méthode de la classe 'Vecteur'.*

2- *L'expression : V1+V2+V1+V2 est évaluée en tenant compte la règle de priorité habituelles de l'opérateur '+' et même chose les autres opérateurs (*, /,-...).*

Exercice VI.1

Développer deux méthode amies permettent de surdéfinit l'opérateur ' >' pour les classes *'Vecteur' et* 'Matrice' (développées précédemment dans les *Exemples IV.1, IV.2et IV.3).*
Tester ensuite l'opérateur surdéfinit '>' dans la fiction 'main'.
Remarque : on constate que : V1 >V2 si ||V1||>||V1|| et même chose pour les matrices.

Solution d'Exercice VI.1

La solution d'exercice est organisée dans les cinq étapes suivantes.

1- Tous d'abord, on doit ajouter la déclaration de l'amitié dans les définitions de deux classes 'Vecteur' et 'Matrice' comme suit :

1-a Classe 'Vecteur'

```
    cout << "]"<<endl;
}

friend void Produit_M_V (Vecteur, Matrice );
// l'opérateur + est défini comme méthode amie
friend Vecteur operator+(Vecteur, Vecteur);
// l'opérateur > est défini comme méthode amie
friend bool operator>(Vecteur, Vecteur);
private:
int v_L=0 ;
int V[10];
};
```

1-b Classe 'Matrice'

```
}
friend void Produit_M_V (Vecteur, Matrice );
friend bool operator>(Matrice, Matrice);
private:
int m_L=0;
int M[10][10];

};
```

Ensuite, on ajoute l'implémentation de deux méthodes amies permettent de surdéfini l'opétateur '>' pour les deux classe 'Vecteur' et 'Matrice'

2- Surdéfinition de l'opérateur > pour la classe 'Vecteur'

```
/*surchage d'un opérateur '>' pour la calsse 'Vecteur'*/
bool operator>(Vecteur vec1, Vecteur vec2)
{
    double module_v1=0, module_v2=0;
    // calcul ||vec1||
        for(int i=0;i<(vec1.v_L); i++)
        {
```

```
            module_v1+= vec1.V[i]*vec1.V[i];
        }
    // calcul ||vec2||
        for(int i=0;i<(vec2.v_L); i++)
        {
            module_v2+= vec2.V[i]*vec2.V[i];
        }

     if (module_v1 >module_v2)
     { return 1;}
     else
     {return 0;}
}
```

3- Surdéfinition de l'opérateur > pour la classe 'Matrice

```
152  /*surchage d'un opérateur '>' pour la calsse 'Matrice'*/
153  bool operator>(Matrice M1, Matrice M2)
154 {
155      double module_m1=0, module_m2=0;
156      // calcul ||M1||
157          for(int i=0;i<(M1.m_L); i++)
158          {
159              for(int j=0;j<(M1.m_L); j++)
160              {
161                  module_m1+= M1.M[i][j]*M1.M[i][j];
162              }
163          }
164      // calcul ||M2||
165          for(int i=0;i<(M2.m_L); i++)
166          {
167              for(int j=0;j<(M2.m_L); j++)
168              {
169                  module_m2+= M2.M[i][j]*M2.M[i][j];
170              }
171
172          }
173       if (module_m1 >module_m2)
174       { return 1;}
175       else
176       {return 0;}
177 }
```

4- Test de la surdéfinition de l'opérateur > pour la classe 'Vecteur' dans la fonction main

```
int main()
{
Vecteur V1(3);
V1.Saisie_V();
Vecteur V2(5);
V2.Saisie_V();
// Teste de l'opérateur > surdéfinit pour la classe Vecteur

if (V1>V2)
{cout << " Oui"<< endl;}
else
{cout << " NOn"<< endl;}
return 0;
}
```

Résultat d'éxécution

```
Saisie du vecteur
 Vecteur[0]= ? 2
 Vecteur[1]= ? 2
 Vecteur[2]= ? 2
Saisie du vecteur
 Vecteur[0]= ? 5
 Vecteur[1]= ? 5
 Vecteur[2]= ? 5
 Vecteur[3]= ? 5
 Vecteur[4]= ? 5
 NOn
```

5- Test de la surdéfinition de l'opérateur > pour la classe 'Matrice' dans la fonction main

```
int main()
{
Matrice M1(3);// 3X3
M1.Saisie_M();
Matrice M2(2);// 2X2
M2.Saisie_M();
// Teste de l'opérateur > surdéfinit pour la classe Matrice
if (M1>M2)
{cout << " Oui"<< endl;}
else
{cout << " NOn"<< endl;}
return 0;
}
```

Résultat d'éxécution

```
saisie d'une matrice
 Matrice [0][0]= ? 1
 Matrice [0][1]= ? 1
 Matrice [0][2]= ? 1
 Matrice [1][0]= ? 1
 Matrice [1][1]= ? 1
 Matrice [1][2]= ? 1
 Matrice [2][0]= ? 1
 Matrice [2][1]= ? 1
 Matrice [2][2]= ? 1
saisie d'une matrice
 Matrice [0][0]= ? 2
 Matrice [0][1]= ? 2
 Matrice [1][0]= ? 2
 Matrice [1][1]= ? 2
 NOn
```

IV.4 Patrons de fonctions et patrons de classes

La notion de 'patron' (template en anglais) permet d'écrire une seule définition pour ensemble des fonctions/classes, c à d la définition d'un patron permet au compilateur d'instancier (créer) selon les types des arguments spécifiés (int, float, char….) plusieurs fonctions ayant le même algorithme. Autrement dit, un patron de fonctions permet de générer (instancier) une famille de fonctions/classes à partir d'une seule définition.

IV.4.1 Exemple introductif

Supposant que nous allons développer une fonction **Maxi** permet de trouver le maximum de deux objets tel que objet peut être l'un des types suivants : 'float', 'int', 'Vecteur' ou 'Matrice'. Dans ce cas-là on a besoin d'une déclaration et algorithme standards permettent de trouver le maximum indépendamment des types d'objet susmentionnés. Dans cette situation la notion de **patron de fonctions** peut intervenir pour remplir cette tâche.

IV.4.2 Syntaxe de la déclaration d'un patron de fonctions

template < **typename***T* > *T nom_de_la fonction* (*T parm* 1, *T parm* 2… *T parm n*)

{

// *code source du patron defonctions*

}

Où : 'template' et 'typename' sont des mots clés, T est paramètre prend la place du type des arguments de la fonction (int, double, char, classe, struct…etc.).

Remarque: il possible d'utiliser le mot '*classe'* à la place de mot '*typename'*.

Exemple IV.4

Le programme suivant montre comment implémenter le patron de fonctions **Maxi** abordé dans le paragraphe IV.4.1.

```
template <typename T> T Maxi (T parm1, T parm2)
{
 if (parm1>parm2) { return parm1;}
 else {return parm2;}
}
```

Le patron 'Maxi' utilise l'opératuer '>' ; l'utilsation de ce patron avec d'autre classes (différentes à 'Vecteur' et 'Matrice') n'est pas possible que avec la surdifinition de l'opérateur > pour chacune.

Ainsi le code complet est résumé comme suite :

```
  1  #include <iostream>
  2  #include <string>
  3  using namespace std;
  4  class Matrice;/*la déclaration de la classe 'Matrice' est
  6
  7  class Vecteur
  8  {
 51  class Matrice
 52  {
 94
 95  //Méthode amie calcule et affiche le produit: Matrice X Vecteur
 96    void Produit_M_V (Vecteur Vec, Matrice Mat )
 97    {
112  /*surchage d'un opérateur+ pour la calsse 'Vecteur'*/
113  Vecteur operator+(Vecteur vec1, Vecteur vec2)
114  {
128
```

```
129  /*surchage d'un opérateur '>' pour la calsse 'Vecteur'*/
130  bool operator>(Vecteur vec1, Vecteur vec2)
131 {
149
150  /*surchage d'un opérateur '>' pour la calsse 'Matrice'*/
151  bool operator>(Matrice M1, Matrice M2)
152 {
176
177  template <typename T> T Maxi (T parm1, T parm2)
178 {
179   if (parm1>parm2) { return parm1;}
180   else {return parm2;}
181 }
```

Le Programme suivant test le patron 'Maxi' pour différent types d'arguments.

```
183  int main()
184 {
185  float k=18.5, p=25.6;// varriables type float
186  int x=10, y=20; // varriables type int
187  Matrice M1(2);// objet1 de la classe Matrice 2X2
188  M1.Saisie_M();
189  Matrice M2(2);// objet2 de la classe Matrice 2X2
190  M2.Saisie_M();
191
192  Vecteur V1(3);//objet1 de la classe 'Vecteur' 1X3
193  V1.Saisie_V();
194  Vecteur V2(3);//objet2 de la classe'Vecteur' 1X3
195  V2.Saisie_V();
196  cout << " Maxi (k,p)= "<< Maxi (k,p)<<endl;
197  cout << " Maxi (x,y)= "<< Maxi (x,y)<<endl;
198  Matrice M3(2);
199  M3= Maxi (M1,M2);
200  cout << " Maxi (M1,M2)= "<<endl;
201  M3.Affiche_M();
202  Vecteur V3(3);
203  V3= Maxi (V1,V2);
204  cout << " Maxi (V1,V2)= "<<endl;
205  V3.Affiche_V();
206  return 0;
207 }
```

Resultat d'éxécution :

```
saisie d'une matrice
 Matrice [0][0]= ? 1
 Matrice [0][1]= ? 1
 Matrice [1][0]= ? 1
 Matrice [1][1]= ? 1
saisie d'une matrice
 Matrice [0][0]= ? 2
 Matrice [0][1]= ? 2
 Matrice [1][0]= ? 2
 Matrice [1][1]= ? 2
Saisie du vecteur
 Vecteur[0]= ? 4
 Vecteur[1]= ? 4
 Vecteur[2]= ? 4
Saisie du vecteur
 Vecteur[0]= ? 5
 Vecteur[1]= ? 5
 Vecteur[2]= ? 5
 Maxi (k,p)= 25.6
 Maxi (x,y)= 20
 Maxi (M1,M2)=
Matrice =
2       2
2       2
 Maxi (V1,V2)=
Vecteur = [ 5 5 5 ]
```

Remarques importantes:

1- *Dans le programme précédent : les appels au patron Maxi sont traduits comme suit :*
   - *Maxi (k, p)// Maxi (float, float).*
   - *Maxi (x,y)// Maxi (int, int).*
   - *Maxi (M1,M2)// Maxi (class Matrice, class Matrice).*
   - *Maxi (V1,V2)// Maxi (class Vecteur, class Vecteur) .*
2- *Il possible d'utiliser le patron avec d'autre type de variables définis (ordinaires) telles que (double, unsigned int, ...).*
3- *Il est nécessaire de prendre en considérations les cas où l'argument est de type char, pointeur ou référence car la comparaison sera faite* ***sur les adresses pas sur leurs contenus***.

IV.4.3 Syntaxe de la déclaration d'un patron de fonctions avec plusieurs parmetres de type

template < **typename** *T*, **typename** *U* > *T nom_de_la fonction* (*T parm* 1, *U parm* 2… *T parm n*)
{
// *code source du patron de fonctions.*
}

Exemple IV. 5

Dans le programme suivant le patron 'somme' a trois types d'arguments (T, U et R) le retour de patron est toujours de type T spécifié par l'utilisateur.

Dans le premier test (ligne 13), on a spécifié T comme float, U comme int et R comme unsigned int. Dans ce cas-là le retour de patron 'somme' est forcément de type 'float'. En revanche, dans le deuxième test (ligne 13) le retour de patron est de type int (la raison que le résultat de la somme est est 8 pas 8.5).

```
1   #include <iostream>
2
3   using namespace std;
4
5   template <typename T, typename U, typename R> T somme (T parm1, U parm2, R parm3)
6   {
7    return parm1+parm2+parm3 ;
8   }
9
10  int main()
11  {
12  float p=1.5; int q=2; unsigned int s=5;
13  cout << "p+q+s= " << somme (p,q,s )<<endl;
14  cout << "q+s+p= " << somme (q,s,p )<<endl;
15  return 0;
16  }
```

Résultat d'éxécution

```
p+q+s= 8.5
q+s+p= 8
```

Exercice IV.2

Développer un patron de fonctions nommé '*find_zero*' permet de déterminer le nombre de zéros dans un tableau statique **de taille et de type quelconques**.

Solution d'Exercice IV.2

```
#include <iostream>
using namespace std;
template <typename T> int find_zero (T* Tab, int taille)
{
int N_zero=0;
for (int i=0; i<taille; i++)
{
if(Tab[i]== 0) {N_zero++;}
}
return N_zero;
}

int main()
{
float tab1[6]={4,10,1988,0,19,0};
int   tab2[8]={0,2,10,0,0,0,7,0};
char  tab3[5]={'a','0','d','e',0};
cout << "find_zero (tab1,  6) = " << find_zero (tab1,6)<<endl;
cout << "find_zero (tab2,  8) = " << find_zero (tab2,8)<<endl;
cout << "find_zero (tab3,  5) = " << find_zero (tab3,5)<<endl;
return 0;
}
```

Résultat d'éxécution

```
find_zero (tab1,  6) = 2
find_zero (tab2,  8) = 5
find_zero (tab3,  5) = 1
```

Dans le programme précédent, *Exercice III.4*, le patron '*find_zero*' instancier (générér) des fonctions dans lesquelles le type de prmeir parametre est standars T* (fixé par l'utilisateur) tandis que le type de retour et le type de deuxième parametre sont imposés par le patron (toujours de type int).

IV.4.4 Surcharge d'un patron de fonctions

Il est possible de surcharger (surdéfinir) un patron de fonctions comme toute fonction classique (voir surchage d'une fonction dans chapitre II ,).

Le programme suivant donne un exemple de la surcharge de patron '*find_zero*' développé précédement dans *Exercice IV.2*.

```
#include <iostream>
using namespace std;
//patron 'find_zero'
/*ce patron est caractérisé par deux parametres, il permet de
trouver le nombre de zéros dans un tableau statique de taille
et de type quelconques*/
```

```
7  template <typename T> int find_zero (T* Tab, int taille)
8  {
9  int N_zero=0;
10 for (int i=0; i<taille; i++)
11 {
12 if(Tab[i]== 0) {N_zero++;}
13 }
14 return N_zero;
15 }
16 // Surcharge de patron 'find_zero'
17 /*Ce patron est caractérisé par trois parametres. Il réalise
18  en plus de la fonction du patron précédent (à deux parametres)
19 le remplacement de tous les zéros du tableau par un nodmbre
20 entrier spécifié par l'utilisateur */
21
22 template <typename T> int find_zero (T* Tab, int taille, int c)
23 {
24 int N_zero=0;
25 for (int i=0; i<taille; i++)
26 {
27 if(Tab[i]== 0) {N_zero++; Tab[i]== c;}
28 }
29 return N_zero;
30 }
31 int main()
32 {
33 int   tab2[8]={0,2,10,0,0,0,7,0};
34 cout << "find_zero (tab2,  8) = " << find_zero (tab2,8)<<endl;
35 cout << "find_zero (tab2,  8, 1) = " << find_zero (tab2,8,1)<<endl;
36
37 return 0;
38 }
```

Résultat d'exécution

```
find_zero (tab2,  8) = 5
find_zero (tab2,  8, 1) = 5
```

Exercice IV.3

Surcharger le patron *'somme'* développé précédemment dans *Exemple III.16* pour qu'il puisse utiliser pour calculer la somme d'un tableau statique de taille et de type quelconques.

Solution d'Exercice IV.3

```
1  #include <iostream>
2  using namespace std;
3  // Patron somme à trois paramètres
4  template <typename T, typename U, typename R> T somme (T parm1, U parm2, R parm3)
5  {
6   return parm1+parm2+parm3 ;
7  }
8  // Patron somme surchargé; à deux paramètres (pointeur et int)
9  template <typename T> T somme (T* tab, int taille)
10 {
11  T somme=0;
12  for (int i=0; i<taille;i++ )
13  {
14   somme += tab[i];
```

```
15      }
16      return somme ;
17  }
```

Test du programme

```
int main()
{
float p=1.5; int q=2; unsigned int s=5;
int   tab1[6]={4,10,1988,19,10,2020};// tab int
float tab2[6]={4.2,10.2,1988.2,19.2,10.2,2020.2};// tab float
cout << "p+q+s= " << somme (p,q,s )<<endl; // patron à trois parametres
cout << "somme tab1= " << somme (tab1,6 )<<endl;//patron à deux parametres T comme int
cout << "somme tab2= " << somme (tab2,6 )<<endl;//patron à deux parametres T comme flaot
return 0;
}
```

Résulta d'exécution

```
p+q+s= 8.5
somme tab1= 4051
somme tab2= 4052.2
```