

---

# **Chapitre 12**

## **Surcharge Des Opérateurs**

## I- Introduction

Le langage C++ autorise le programmeur à étendre la signification d'opérateurs tels que l'addition (+), la soustraction (-), la multiplication (\*), la division (/), le ET logique (&) etc...

### Exemple:

On considère la classe *vecteur* et on surdéfinit l'opérateur somme (+) qui permettra d'écrire dans un programme:

```
vecteur v1, v2, v3;  
v3 = v2 + v1;
```

```

#include <iostream>
using namespace std; // Classe vecteur : surcharge de l'opérateur +
class vecteur
{
private:
float x,y;
public:
vecteur(float abs,float ord);
void affiche();
// surcharge de l'opérateur somme, on passe un paramètre vecteur
// la fonction retourne un vecteur
vecteur operator+(vecteur v);
};

vecteur::vecteur(float abs =0,float ord = 0)
{
x = abs; y = ord;
}

void vecteur::affiche()
{
cout<<"x = "<< x <<" y = "<< y <<"\n";
}

vecteur vecteur::operator+(vecteur v)
{
vecteur res;
res.x = v.x + x; res.y = v.y + y;
return res;
}

void main()
{
vecteur a(2,6),b(4,8),c,d,e,f;
c = a + b; c.affiche();
d = a.operator+(b); d.affiche();
e = b.operator+(a); e.affiche();
f = a + b + c; f.affiche();
}

```

### Exercice 1:

Ajouter une fonction membre de prototype

**float operator\*(vecteur v)**

permettant de créer l'opérateur « produit scalaire », c'est à dire de donner une signification à l'opération suivante:

**vecteur v1, v2;**

**float prod\_scal;**

**prod\_scal = v1 \* v2;**

### Exercice 2:

Ajouter une fonction membre de prototype

**vecteur operator\*(float)**

permettant de donner une signification au produit d'un réel et d'un vecteur selon le modèle suivant :

**vecteur v1,v2;**

**float h;**

**v2 = v1 \* h ; // homothétie**

---

### Exercice 3:

Sans modifier la fonction précédente, essayer l'opération suivante et conclure.

```
vecteur v1,v2;  
float h;  
v2 = h * v1; // homothétie
```

- - ✓ Cet appel conduit à une erreur de compilation.
  - ✓ L'opérateur ainsi créé, n'est donc pas symétrique.
  - ✓ Il faudrait disposer de la notion de « fonction amie » pour le rendre symétrique.

---

## 2- Application: Utilisation d'une bibliothèque

**C++ possède une classe « complex », dont le prototype est déclaré dans le fichier `complex.h`.**

▪

**Voici une partie de ce prototype:**

```

class complex
{
private:
double re,im; // partie réelle et imaginaire d'un nombre complexe
public:
complex(double reel, double imaginaire = 0); // constructeur
// complex manipulations
double real(complex); // retourne la partie réelle
double imag(complex); // retourne la partie imaginaire
complex conj(complex); // the complex conjugate
double norm(complex); // the square of the magnitude
double arg(complex); // the angle in radians
complex polar(double mag, double angle=0); // Create a complex object given polar coordinates
complex operator+(complex);
friend complex operator+(double, complex); // donnent un sens à : « complex + double » et « double + complex »
friend complex operator+(complex, double); // la notion de « fonction amie » sera étudiée au prochain chapitre
complex operator-(complex); // idem avec la soustraction
friend complex operator-(double, complex);
friend complex operator-(complex, double);
complex operator*(complex);
friend complex operator*(complex, double); // idem avec la multiplication
friend complex operator*(double, complex);
complex operator/(complex);
friend complex operator/(complex, double); // idem avec la division
friend complex operator/(double, complex);
int operator==(complex); // retourne 1 si égalité
int operator!=(complex); // retourne 1 si non égalité
complex operator-(); // opposé du vecteur
// Complex stream I/O
// permet d'utiliser cout avec un complexe
ostream operator<<(ostream, complex);
// permet d'utiliser cin avec un complexe
istream operator>>(istream, complex);
};

```

## Remarques Générales:

➤ **Pratiquement tous les opérateurs peuvent être surdéfinis:**

+ - \* / = ++ -- new delete [] -> & | ^ && || % << >> etc ...

➤ **Il faut se limiter aux opérateurs existants.**

▪ ➤ **Les règles d'associativité et de priorité sont maintenues.**

➤ **Il n'en est pas de même pour la commutativité.**

➤ **L'opérateur = peut-être redéfini.**

*Si la classe contient des données dynamiques, il faut impérativement surcharger l'opérateur=.*



## Exemple:

**La surcharge de l'opérateur d'affectation de la classe liste vue précédemment est donné par:**

```
liste& liste::operator=(const liste &obj)
{
    taille=obj.taille;
    delete [ ] adr;
    adr= new float[taille];
    for(int i=0;i<taille;i++)
        adr[i]=ob.adr[i];
    cout<<"surcharge de l'opérateur ="<<endl;
    return *this;
}
```

---

## Conclusion:

✓ Une classe qui présente une donnée membre allouée dynamiquement doit toujours posséder au minimum:

- un constructeur,
- ➤ un constructeur par copie,
- un destructeur,
- la surcharge de l'opérateur =.

✓ Une classe qui possède ces propriétés est appelée « classe canonique ».

---

# **Chapitre 13**

## **Fonctions amies**

- 
- ✓ **Nous avons vu qu'une classe avait généralement des membres privés, et que ceux-ci n'étaient pas accessibles par des fonctions non membres.**
  - ✓ **Cette restriction peut sembler lourde, mais elle est à la base même de la protection des données qui fait une grande partie de la puissance de la programmation par objets en général, et de C++ en particulier.**
  - ✓ **Dans certains cas, cependant, on souhaite pouvoir utiliser une fonction qui puisse accéder aux membres privés d'une classe, sans toutefois nécessairement disposer d'une instance de cette classe par laquelle l'appeler.**

---

✓ Le langage C++ fournit ce qu'on appelle les fonctions *amies* comme solution de ce problème.

✓ Il existe plusieurs situations d'amitié:

- - Une fonction indépendante est amie d'une ou de plusieurs classes.
  - Une ou plusieurs fonctions membres d'une classe sont amies d'une autre classe.

## I- Fonction indépendante amie d'une classe

### Exemple :

Dans l'exemple ci-dessous, la fonction *coïncide* est *amie* de la classe *point*. C'est une fonction ordinaire qui peut manipuler les membres privés de la classe *point*.

```
#include <iostream> // fonction indépendante, amie d'une classe
using namespace std;
class point
{
private :
int x,y;
public:
point(int abs=0,int ord=0){x=abs;y=ord;}
//déclaration de la fonction amie
friend bool coincide(point,point);
};
bool coincide(point p,point q)
{
if ((p.x==q.x) && (p.y==q.y))          return true;
return false;
}
void main()
{point a(4,0),b(4),c;
if(coincide(a,b))      |cout<<"a coincide avec b\n";
else cout<<"a est différent de b\n";
if(coincide(a,c))      cout<<"a coincide avec c\n";
else cout<<"a est différent de c\n";
}
```

## **II- Les autres situations d'amitié :**

### **Situation 1:**

- ✓ On souhaite parfois qu'une méthode d'une classe puisse accéder aux parties privées d'une autre classe.
- ✓ Pour cela, il suffit de déclarer la méthode friend également, en utilisant son nom complet (nom de classe suivi de :: et du nom de la méthode).

### **Par exemple :**

Dans la situation suivante, la fonction `fm_de_C1`, fonction membre de la classe `C1`, a accès aux membres privés de la classe `C2`:

```

class C2
{
    // partie privée
    .....
    ....
    // partie publique
    friend int C1::fm_de_ C21(char, C2);
};

class C1
{
    .....
    ....
    int fm_de_ C21(char, C2);
};

int C1::fm_de_ C21(char c, C2 t)
{
    ....
} // on pourra trouver ici une invocation des membres privés de l'objet t

```

✓ Si toutes les fonctions membres de la classe C1 étaient amies de la classe C2, on déclarerait directement dans la partie publique de la classe C2: **friend class C1;**



## Situation 2:

Dans la situation ci-dessous, la fonction `f_anonyme` a accès aux membres privés des classes `C2` et `C1`:

```
class C2
{
    // partie privée
    .....
    .....
    // partie publique
    friend void f_anonyme(C2, C1);
};

class C1
{
    // partie privée
    .....
    .....
    // partie publique
    friend void f_anonyme(C2, C1);
};

void f_anonyme(C2 t2, C1 t1)
{
    ...
} // on pourra trouver ici une invocation des membres privés des objets t2 et t1.
```

---

## III- Application à la surcharge des opérateurs

### Exemple:

- ✓ On reprend l'exemple permettant de surcharger l'opérateur `+` pour l'addition de deux vecteurs.
- ✓ On crée, cette fois-ci, une fonction amie de la classe vecteur.

```

#include <iostream> // Classe vecteur
Using namespace std; // Surcharge de l'opérateur + par une fonction AMIE
class vecteur
{
private:
float x,y;
public: vecteur(float,float);
void affiche();
friend vecteur operator+(vecteur, vecteur);
};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs; y=ord;}
void vecteur::affiche()
{cout<<"x = "<< x << " y = " << y << "\n";}
vecteur operator+(vecteur v, vecteur w)
{
vecteur res;
res.x = v.x + w.x;
res.y = v.y + w.y;
return res;
}

void main()
{vecteur a(2,6),b(4,8),c,d;
c = a + b;
c.affiche();
d = a + b + c;
d.affiche();
}

```

---

## Exercice 1 :

Soit la classe *vecteur*, en utilisant la propriété de surcharge des fonctions du C++, créer

- une fonction membre de la classe *vecteur* de prototype  
 $\text{float vecteur::operator*}(\text{vecteur});$   
qui retourne le produit scalaire de deux vecteurs
- une fonction membre de la classe *vecteur* de prototype  
 $\text{vecteur vecteur::operator*}(\text{float});$   
qui retourne le vecteur produit d'un vecteur et d'un réel (donne une signification à  $v2 = v1 * h;$ )
- une fonction AMIE de la classe *vecteur* de prototype  
 $\text{vecteur operator*}(\text{float}, \text{vecteur});$   
qui retourne le vecteur produit d'un réel et d'un vecteur (donne une signification à  $v2 = h * v1;$ )

- 
- On doit donc pouvoir écrire dans le programme:

```
vecteur v1, v2, v3, v4;  
float h, p;  
p = v1 * v2;  
v3 = h * v1;  
v4 = v1 * h;
```

### Remarque:

On aurait pu remplacer la fonction membre de prototype `vecteur vecteur::operator*(float);` par une fonction *amie* de prototype `vecteur operator*(vecteur, float);`

### Exercice 2:

Étudier le listing du fichier d'en-tête `complex.h` et justifier tous les prototypes des fonctions.

## Exemples:

La surcharge des opérateurs << et >> de la classe liste vue predement est donné par:

```
friend ostream& operator<<(ostream &str, liste &);
friend istream& operator>>(istream &, liste &);
ostream& operator<<(ostream &str, liste & p)
{ str<<"[";
  for(int i=0;i<p.taille;i++)
      str<<" "<<p.adr[i];

  str<<" ]";
  return str;
}
istream& operator>>(istream &istr, liste & p)
{ cout<<"Entrer le nombre d'elt de votre liste"<<endl;
  istr>>p.taille;
  cout<<"Entrer votre liste"<<endl;
  for(int i=0;i<p.taille;i++)
      istr>>p.adr[i];

  return istr;
}
```