# Chapitre 5 : Opérateurs

## I52 Programmation Orientée objet — C++

Valérie Gillot

Septembre 2019

# Plan du chapitre

Section 1

# Surcharge des opérateurs

- Principe
- Par une fonction membre
- Par une fonction non membre

# Principe

## Surcharge

La surcharge des opérateurs en C++ permet de redéfinir la sémantique des opérateurs

- Soit pour l'étendre à des objets
- Soit pour modifier l'effet d'opérateurs prédéfinis sur des objets

## Règles de surcharge des opérateurs

1. Il n'est pas possible d'inventer de nouveaux opérateurs

# Principe

## Surcharge

La surcharge des opérateurs en C++ permet de redéfinir la sémantique des opérateurs

- Soit pour l'étendre à des objets
- Soit pour modifier l'effet d'opérateurs prédéfinis sur des objets

## Règles de surcharge des opérateurs

1. Il n'est pas possible d'inventer de nouveaux opérateurs
2. Seuls les opérateurs connus du compilateur peuvent être surchargés
3. Tous les opérateurs de C++ ne peuvent pas être surchargés, entre autres les opérateurs :

   `.` `.*` `::` `?` `:` `sizeof`

4. On ne peut surcharger un opérateur appliqué exclusivement à des types standards : nécessité d'au moins un opérande de type classe

# Principe

## Surcharge

La surcharge des opérateurs en C++ permet de redéfinir la sémantique des opérateurs

- Soit pour l'étendre à des objets
- Soit pour modifier l'effet d'opérateurs prédéfinis sur des objets

## Règles de surcharge des opérateurs

1. Il n'est pas possible d'inventer de nouveaux opérateurs
2. Seuls les opérateurs connus du compilateur peuvent être surchargés
3. Tous les opérateurs de C++ ne peuvent pas être surchargés, entre autres les opérateurs :

   `.` `.*` `::` `?:` `sizeof`

4. On ne peut surcharger un opérateur appliqué exclusivement à des types standards : nécessité d'au moins un opérande de type classe

# Principe

## Surcharge

La surcharge des opérateurs en C++ permet de redéfinir la sémantique des opérateurs

- Soit pour l'étendre à des objets
- Soit pour modifier l'effet d'opérateurs prédéfinis sur des objets

## Règles de surcharge des opérateurs

1. Il n'est pas possible d'inventer de nouveaux opérateurs
2. Seuls les opérateurs connus du compilateur peuvent être surchargés
3. Tous les opérateurs de C++ ne peuvent pas être surchargés, entre autres les opérateurs :

   `.` `.*` `::` `?:` `sizeof`
4. On ne peut surcharger un opérateur appliqué exclusivement à des types standards : nécessité d'au moins un opérande de type classe

# Principe

## Syntaxe et sémantique

La surcharge de l'opérateur ♣ se fait par la définition d'une fonction

`type operator ♣(arguments)`

- soit par une fonction membre (méthode)
- soit par une fonction non membre

## Propriétés de l'opérateur surchargé

Après la surcharge :

- Conservation de la pluralité, de la priorité et associativité.
- Perte éventuelle de la commutativité et liens sémantiques avec les autres opérateurs

# Principe

## Syntaxe et sémantique

La surcharge de l'opérateur ♣ se fait par la définition d'une fonction

`type operator ♣(arguments)`

- soit par une fonction membre (méthode)
- soit par une fonction non membre

## Propriétés de l'opérateur surchargé

Après la surcharge :

- Conservation de la pluralité, de la priorité et associativité.
- Perte éventuelle de la commutativité et liens sémantiques avec les autres opérateurs

# Principe

## Syntaxe et sémantique

La surcharge de l'opérateur ♣ se fait par la définition d'une fonction

`type operator ♣(arguments)`

- soit par une fonction membre (méthode)
- soit par une fonction non membre

## Propriétés de l'opérateur surchargé

Après la surcharge :

- Conservation de la pluralité, de la priorité et associativité.
- Perte éventuelle de la commutativité et liens sémantiques avec les autres opérateurs

# Principe

## Syntaxe et sémantique

La surcharge de l'opérateur ♣ se fait par la définition d'une fonction

`type operator ♣(arguments)`

- soit par une fonction membre (méthode)
- soit par une fonction non membre

## Propriétés de l'opérateur surchargé

Après la surcharge :

- Conservation de la pluralité, de la priorité et associativité.
- Perte éventuelle de la commutativité et liens sémantiques avec les autres opérateurs

# Surcharge par une fonction membre

## Déclaration

Si la surcharge est une méthode de la classe,

- le premier opérande est l'objet courant
- il y a un paramètre de moins que la pluralité de l'opérateur

## Appel : cas opérateurs unaires ou binaires

- Opérateur unaire : `♣obj` ou `obj♣` est équivalent à `obj.operator♣()`
- Opérateur binaire : `obj1 ♣ obj2` est équivalent à `obj1.operator♣(obj2)`

# Surcharge par une fonction membre : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une
      fonction membre
  Point operator+(Point);
  };
```

## point.cc (C)

```
Point Point::operator+(Point P)
{
return Point(x+P.x,y+P.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R = P.operator+(Q)
```

# Surcharge par une fonction membre : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une
      fonction membre
  Point operator+(Point);
  };
```

## point.cc (C)

```
Point Point::operator+(Point P)
{
return Point(x+P.x,y+P.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R = P.operator+(Q)
```

# Surcharge par une fonction membre : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une
      fonction membre
  Point operator+(Point);
  };
```

## point.cc (C)

```
Point Point::operator+(Point P)
{
return Point(x+P.x,y+P.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R = P.operator+(Q)
```

# Surcharge par une fonction non membre

## Déclaration

Si la surcharge n'est pas une méthode de la classe, il y a autant de paramètres que la pluralité de l'opérateur

## Appel : cas opérateurs unaires ou binaires

- Opérateur unaire : ♣`obj` ou `obj`♣ est équivalent à `operator`♣`(obj)`
- Opérateur binaire : `obj1` ♣ `obj2` est équivalent à `operator`♣`(obj1, obj2)`

# Surcharge par une fonction amie : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une amie
  friend Point operator+(Point,Point);
  };
```

## point.cc (C)

```
Point operator+(Point P, Point Q)
{
return Point(P.x+Q.x,P.y+Q.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R =operator+(P,Q)
```

# Surcharge par une fonction amie : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une amie
  friend Point operator+(Point,Point);
  };
```

## point.cc (C)

```
Point operator+(Point P, Point Q)
{
return Point(P.x+Q.x,P.y+Q.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R =operator+(P,Q)
```

# Surcharge par une fonction amie : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de + par une amie
  friend Point operator+(Point,Point);
  };
```

## point.cc (C)

```
Point operator+(Point P, Point Q)
{
return Point(P.x+Q.x,P.y+Q.y);
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
R = P + Q;// R =operator+(P,Q)
```

# Surcharge par une fonction non membre : usage

## Fonction amie de la classe

1. La déclaration amicale permet d'accéder aux attributs, si on déclare la fonction en dehors de la classe il faut utiliser les accesseurs pour accéder aux attributs privés
2. on ne peut définir les deux surcharges (membre et non membre) en même temps, P+Q serait ambiguë pour le compilateur
3. la surcharge d'un opérateur binaire par une fonction non membre est obligatoire, si le premier opérande n'est pas une instance de la classe.

# Surcharge par une fonction non membre : usage

## Fonction amie de la classe

1. La déclaration amicale permet d'accéder aux attributs, si on déclare la fonction en dehors de la classe il faut utiliser les accesseurs pour accéder aux attributs privés
2. on ne peut définir les deux surcharges (membre et non membre) en même temps, `P+Q` serait ambiguë pour le compilateur
3. la surcharge d'un opérateur binaire par une fonction non membre est obligatoire, si le premier opérande n'est pas une instance de la classe.

# Surcharge par une fonction non membre : usage

## Fonction amie de la classe

1. La déclaration amicale permet d'accéder aux attributs, si on déclare la fonction en dehors de la classe il faut utiliser les accesseurs pour accéder aux attributs privés
2. on ne peut définir les deux surcharges (membre et non membre) en même temps, `P+Q` serait ambiguë pour le compilateur
3. la surcharge d'un opérateur binaire par une fonction non membre est obligatoire, si le premier opérande n'est pas une instance de la classe.

Section 2

# Opérateurs remarquables

- Affectation
- Flux
- Indexation

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## La surcharge implicite

La surcharge de l'opérateur d'affectation est synthétisé par le compilateur si le programmeur ne le fait pas. Les attributs sont alors affectés champs à champs.

## La surcharge explicite

La surcharge synthétisée par le compilateur pose des problème dans le cas d'attributs alloués dynamiquement. Dans ce cas, la surcharge est nécessaire, elle comprend alors pour `obj1 = obj2`

1. la désallocation de l'objet courant `obj1`
2. de l'allocation de l'objet courant de même taille que obj2
3. copie de `obj2` dans l'objet courant
4. retourne l'objet courant modifié

## Fonction membre

L'affectation est surchargée par une fonction membre

# Surcharge l'opérateur d'affectation

## Déclaration

Pour une classe `C` :

```
C& operator=(const C &)
```

## Retour d'une référence

Le retour de la méthode par référence permet de transmettre l'objet courant pour des appels d'affectations successives : `obj1 = obj2 = obj3 = ···`

# Surcharge l'opérateur d'affectation

## Déclaration

Pour une classe `C` :

```
C& operator=(const C &)
```

## Retour d'une référence

Le retour de la méthode par référence permet de transmettre l'objet courant pour des appels d'affectations successives : `obj1 = obj2 = obj3 = ···`

# Surcharge l'opérateur d'affectation : exemple

## tabentier.h (C)

```
class TabEntier{
private :
  int taille;
  int* tab;
public:
  TabEntier();
  TabEntier(int,int*);
  TabEntier(const TabEntier &);
  ~TabEntier();
  //surcharge de l'affectaion
  TabEntier & operator=(const TabEntier&);
};
```

# Surcharge l'opérateur d'affectation : exemple

tabentier.cc (C)

```
TabEntier & TabEntier::operator=(const TabEntier& T){
  if (this != &T){
  //desallocation de l'objet courant
    delete [] tab;
  //allocation de l'objet courant
    taille = T. taille;
    tab = new int[taille];
  //copie de T dans obj courant
    for(int i=0; i<taille ; i++)
      tab[i]= T.tab[i];
    }
   // retour objet courant modifie
   return (*this);
}
```

# Surcharge l'opérateur d'affectation : exemple

## tabentier.cc (C)

```
TabEntier & TabEntier::operator=(const TabEntier& T){
  if (this != &T){
  //desallocation de l'objet courant
    delete [] tab;
  //allocation de l'objet courant
    taille = T. taille;
    tab = new int[taille];
  //copie de T dans obj courant
    for(int i=0; i<taille ; i++)
      tab[i]= T.tab[i];
     }
   // retour objet courant modifie
   return (*this);
}
```

# Synthèse de la surcharge l'opérateur d'affectation

## Exercice

### Point

```
int main()
{
Point P(2,3), Q(4,5), R, T;
R = P;
P = T;
return 0;
}
```

### TabEntier

```
int main()
{
TabEntier U(1,{2}), V
     (2,{3,4}), W;
W = U;
U = V;
return 0;
}
```

1. Pour la classe `Point`, simuler l'exécution en mémoire du programme `Point`
2. Pour la classe `TabEntier`, simuler l'exécution en mémoire du programme `TabEntier`, dans les deux cas
   1. avec synthèse de la surcharge de l'affectation par le compilateur
   2. avec la surcharge explicite de l'affectation par le concepteur de la classe

# Surcharge l'opérateur « : flux sortant

## Opérateur d'insertion dans le flux sortant

L' appel de la surcharge d'insertion dans le flux sortant est de la forme `flux « obj1 « obj2`. Ce n'est pas une fonction membre.

## Déclaration pour une classe `C`

```
friend ostream & operator<<(ostream&, const C &);
```

## Retour d'une référence

Le retour et la transmission du flux par référence permet le modifier pour des appels successifs : `flux « obj1 « obj2« obj3 « ···`

## Définition pour une classe `C`

```
ostream & operator<<(ostream& o, const C & obj){
// insertion dans le flux des attributs de obj
return o;}
```

# Surcharge l'opérateur « : flux sortant

## Opérateur d'insertion dans le flux sortant

L' appel de la surcharge d'insertion dans le flux sortant est de la forme `flux « obj1 « obj2`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend ostream & operator<<(ostream&, const C &);
```

## Retour d'une référence

Le retour et la transmission du flux par référence permet le modifier pour des appels successifs : `flux « obj1 « obj2« obj3 « ···`

## Définition pour une classe C

```
ostream & operator<<(ostream& o, const C & obj){
// insertion dans le flux des attributs de obj
return o;}
```

# Surcharge l'opérateur « : flux sortant

## Opérateur d'insertion dans le flux sortant

L' appel de la surcharge d'insertion dans le flux sortant est de la forme `flux « obj1 « obj2`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend ostream & operator<<(ostream&, const C &);
```

## Retour d'une référence

Le retour et la transmission du flux par référence permet le modifier pour des appels successifs : `flux « obj1 « obj2« obj3 « ···`

## Définition pour une classe C

```
ostream & operator<<(ostream& o, const C & obj){
// insertion dans le flux des attributs de obj
return o;}
```

# Surcharge l'opérateur « : flux sortant

## Opérateur d'insertion dans le flux sortant

L' appel de la surcharge d'insertion dans le flux sortant est de la forme `flux « obj1 « obj2`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend ostream & operator<<(ostream&, const C &);
```

## Retour d'une référence

Le retour et la transmission du flux par référence permet le modifier pour des appels successifs : `flux « obj1 « obj2« obj3 « ···`

## Définition pour une classe C

```
ostream & operator<<(ostream& o, const C & obj){
// insertion dans le flux des attributs de obj
return o;}
```

# Surcharge de « : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de << par une amie
  friend ostream& operator<<(ostream &, const Point &);
  };
```

# Surcharge de « : exemple

point.cc (C)

```
ostream& operator<<(ostream & o, const Point & P)
{
o<<"(" << P.x<< "," << P.y <<")";
return o;
}
```

main.cc (U)

```
Point P(2,3),Q(4,5),R;
cout<< P << Q << R << endl;
```

# Surcharge de « : exemple

## point.cc (C)

```
ostream& operator<<(ostream & o, const Point & P)
{
o<<"(" << P.x<< "," << P.y <<")";
return o;
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
cout<< P << Q << R << endl;
```

# Surcharge de « : exemple

## point.cc (C)

```
ostream& operator<<(ostream & o, const Point & P)
{
o<<"(" << P.x<< "," << P.y <<")";
return o;
}
```

## main.cc (U)

```
Point P(2,3),Q(4,5),R;
cout<< P << Q << R << endl;
```

# Surcharge l'opérateur » : flux entrant

## Opérateur d'insertion dans le flux entrant

L' appel de la surcharge d'insertion dans le flux entrant est de la forme `flux » obj`. Ce n'est pas une fonction membre.

### Déclaration pour une classe C

```
friend istream & operator>>(istream&, C &);
```

### Définition pour une classe C

```
istream & operator>>(istream& i, C & obj){
// insertion du flux vers les attributs de obj
return i;}
```

Notons que l'objet ne peut pas être constant et c'est forcément une référence

# Surcharge l'opérateur » : flux entrant

## Opérateur d'insertion dans le flux entrant

L' appel de la surcharge d'insertion dans le flux entrant est de la forme `flux » obj`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend istream & operator>>(istream&, C &);
```

## Définition pour une classe C

```
istream & operator>>(istream& i, C & obj){
// insertion du flux vers les attributs de obj
return i;}
```

Notons que l'objet ne peut pas être constant et c'est forcément une référence

# Surcharge l'opérateur » : flux entrant

## Opérateur d'insertion dans le flux entrant

L' appel de la surcharge d'insertion dans le flux entrant est de la forme `flux » obj`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend istream & operator>>(istream&, C &);
```

## Définition pour une classe C

```
istream & operator>>(istream& i, C & obj){
// insertion du flux vers les attributs de obj
return i;}
```

Notons que l'objet ne peut pas être constant et c'est forcément une référence

# Surcharge l'opérateur » : flux entrant

## Opérateur d'insertion dans le flux entrant

L' appel de la surcharge d'insertion dans le flux entrant est de la forme `flux » obj`. Ce n'est pas une fonction membre.

## Déclaration pour une classe C

```
friend istream & operator>>(istream&, C &);
```

## Définition pour une classe C

```
istream & operator>>(istream& i, C & obj){
// insertion du flux vers les attributs de obj
return i;}
```

Notons que l'objet ne peut pas être constant et c'est forcément une référence

# Surcharge de » : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point()
    {x=0,y=0;}
  Point(int a, int b)
    {x=a; y=b;}
  //surcharge de >> par une amie
  friend istream& operator>>(istream &, Point &);
  };
```

# Surcharge de » : exemple

point.cc (C)

```
istream& operator>>(istream & i, Point & P)
{
i>>P.x;
i>>P.y;
return i;
}
```

main.cc (U)

```
Point P;
cin >> P;
```

# Surcharge de » : exemple

## point.cc (C)

```
istream& operator>>(istream & i, Point & P)
{
i>>P.x;
i>>P.y;
return i;
}
```

## main.cc (U)

```
Point P;
cin >> P;
```

# Surcharge de » : exemple

## point.cc (C)

```
istream& operator>>(istream & i, Point & P)
{
i>>P.x;
i>>P.y;
return i;
}
```

## main.cc (U)

```
Point P;
cin >> P;
```

# Surcharge l'opérateur [] : indexation

## Opérateur d'indexation : syntaxe

L'opérateur d'indexation doit être surchargé par une fonction membre. Deux prototypes possibles :

1. `type& operator[] (int);`
2. `type operator[] (int)const;`

## Opérateur d'indexation : sémantique

Dans les deux cas, retourne l'élément d'indice donné et

1. autorise sa modification
2. le protège de toute modification

# Surcharge l'opérateur [] : indexation

## Opérateur d'indexation : syntaxe

L'opérateur d'indexation doit être surchargé par une fonction membre. Deux prototypes possibles :

1. `type& operator[] (int);`
2. `type operator[] (int)const;`

## Opérateur d'indexation : sémantique

Dans les deux cas, retourne l'élément d'indice donné et

1. autorise sa modification
2. le protège de toute modification

# Surcharge l'opérateur [] : exemple

## tabreel.h (C)

```
class TabReel{
private :
  int taille;
  float* tab;
public:
    TabReel(int n){
      taille=n;
      tab = new float[taille];}
  //surcharge de l'indexation
  // modification autorisee
  float & operator[](int i)
    {return tab[i];}
  // consultation
  float operator[](int i)const
    {return tab[i];}
};
```

## main.cc (U)

```
TabReel T(5); // construction d'une
    instance tableau de 5 float
...
T[2] = 0 ;// T.operator[](i) = 0
cout<< T[4];// T.operator[](j) = 0
...
void f(TabReel &U, const TabReel V){
...
cout<<U[i];//OK
U[i] = 0 ;//OK
cout<<V[j];//OK
V[j] = 0;//KO
...}
```