- Introduction à la surcharge d'opérateurs
- Surcharge par une fonction membre
- Surcharge par une fonction globale
- Opérateur d'affectation
- Surcharge de ++
- Opérateurs de conversion

SURCHARGE D'OPERATEURS

Introduction à la surcharge d'opérateurs :

Le concepteur d'une classe doit fournir à l'utilisateur de celle ci toute une série d'opérateurs agissant sur les objets de la classe. Ceci permet une syntaxe intuitive de la classe.

Par exemple, il est plus intuitif et plus clair d'additionner deux matrices en surchargeant l'opérateur d'addition et en écrivant :

result = m0 + m1; que d'écrire *matrice_add(result, m0, m1);*

Règles d'utilisation :

- Il faut veiller à respecter l'esprit de l'opérateur. Il faut faire avec les types utilisateurs des opérations identiques à celles que font les opérateurs avec les types prédéfinis.
- La plupart des opérateurs sont surchargeables.
- les opérateurs suivants ne sont pas surchargeables : ***::*** ***.*** ***.**** ***?:*** ***sizeof***
- il n'est pas possible de :
  - changer sa priorité
  - changer son associativité

- changer sa pluralité (unaire, binaire, ternaire)
- créer de nouveaux opérateurs

Quand l'opérateur + (par exemple) est appelé, le compilateur génère un appel à la fonction ***operator+***.

Ainsi, l'instruction ***a = b + c;*** est équivalente aux instructions :

```
a = operator+(b, c);   // fonction globale

a = b.operator+(c);    // fonction membre
```

Les opérateurs = *()* *[]* *->* *new* *delete* ne peuvent être surchargés que comme des fonctions membres.

SURCHARGE PAR UNE FONCTION MEMBRE

Par exemple, la surcharge des opérateurs **+** et **=** par des fonctions membres de la classe *Matrice* s'écrit :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice

class Matrice {  // matrice dim1 x dim2 d'entiers
  public:

    // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);

    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

// ...

void main() {
  Matrice b, c;

  b + c ;     // appel à : b.operator+( c );
  a = b + c;  // appel à : a.operator=(  b.operator+( c ) );
}
```

Une fonction membre (non statique) peut toujours utiliser le pointeur caché *this*.

Dans le code ci dessus, *this* fait référence à l'objet *b* pour l'opérateur + et à l'objet *a* pour

l'opérateur =.

Définition de la fonction membre *operator+* :

```
Matrice Matrice::operator+(const Matrice &c) {
  Matrice m;

  for(int i = 0; i < dim1; i++)
    for(int j = 0; j < dim2; j++)
      m._matrice[i][j] = this->_matrice[i][j] + c._matrice[i][j];
  return m;
}
```

Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable. Une fonction membre renforce l'encapsulation. Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage (sauf l'affectation).

La fonction membre *operator+* peut elle même être surchargée, pour dans l'exemple qui suit, additionner à une matrice un vecteur :

```
class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);
    Matrice operator+(const Vecteur &n2);

    // ...
};

// ...

Matrice b, c;
Vecteur v;

b + c; // appel à b.operator+(c);
b + v; // appel à b.operator+(v); addition entre une matrice
       // et un vecteur
v + b; // appel à v.operator+(b); --> ERREUR si la classe
       // Vecteur n'a pas défini l'addition entre un vecteur et
       // une matrice
```

SURCHARGE PAR UNE FONCTION GLOBALE

Cette façon de procéder est plus adaptée à la surcharge des opérateurs binaires.

En effet, elle permet d'appliquer des conversions implicites au premier membre de l'expression.

Exemple :

```
Class Nombre{
  friend Nombre operator+(const Nombre &, const Nombre &);

  public:
    Nombre(int n = 0) { _nbre = n; }
    //....
  private:
    int _nbre;
};


Nombre operator+(const Nombre &nbr1, const Nombre &nbr2) {
  Nombre n;

  n._nbre = nbr1._nbre + nbr2._nbre;
  return n;
}

void main() {
  Nombre n1(10);

  n1 + 20;  // OK appel à :  operator+( n1, Nombre(20) );
  30 + n1;  // OK appel à :  operator+( Nombre(30) , n1 );
}
```

OPERATEUR D'AFFECTATION

C'est le même problème que pour le constructeur de copie.

Le compilateur C++ construit par défaut un opérateur d'affectation "bête".

L'opérateur d'affectation est obligatoirement une fonction membre et il doit fonctionner correctement dans les deux cas suivants :

```
X x1, x2, x3; // 3 instances de la classe X

x1 = x1;

x1 = x2 = x3;
```

Exemple : Opérateur d'affectation de la classe Matrice :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice

class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    const Matrice &operator=(const Matrice &m);
    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

const Matrice &Matrice::operator=(const Matrice &m) {
  if ( &m != this ) {   // traitement du cas :   x1 = x1
    for(int i = 0; i < dim1; i++)   // copie de la matrice
      for(int j = 0; j < dim2; j++)
        this->_matrice[i][j] = m->_matrice[i][j];
  }
  return *this;  // traitement du cas :    x1 = x2 = x3
}
```

SURCHARGE DE ++ et −−

- **notation préfixée :**
  - fonction membre : **X operator++();**
  - fonction globale : **X operator++(X &);**

- **notation postfixée :**
  - fonction membre : **X operator++(int);**
  - fonction globale : **X operator++(X &, int);**

Exemple :

```
class BigNombre {
  public:
    // ...
    BigNombre operator++();    // prefixe
    BigNombre operator++(int); // postfixe
    // ...
}

// ...

void main() {
  BigNombre n1;

  n1++; // notation postfixéé
  ++n1; // notation préfixée
}
```

OPERATEURS DE CONVERSION

Dans la définition complète d'une classe, il ne faut pas oublier de définir des opérateurs de conversions de types.

Il existe deux types de conversions de types :

- la conversion de type prédéfini (ou défini préalablement) vers le type classe en question. Ceci sera effectué grâce au **constructeur de conversion**.

- la conversion du type classe vers un type prédéfini (ou défini préalablement). Cette conversion sera effectuée par des **fonctions de conversion**.

Constructeur de conversion :

Un constructeur avec un seul argument de type *T* permet de réaliser une conversion d'une variable de type *T* vers un objet de la classe du constructeur.

```
class Nombre {
  public:
    Nombre(int n) { _nbre = n; }
  private:
    int _nbre;
};

void f1(Nombre n) { /* ... */ }

void main() {
  Nombre n = 2; // idem que :  Nombre n(2);

  f1(3); // Appel du constructeur Nombre(int) pour réaliser
         // la conversion de l'entier 3 en un Nombre.
         // Pas d'appel du constructeur de copie
}
```

Dans cet exemple, le constructeur avec un argument permet donc d'effectuer des conversions d'entier en *Nombre.*

Fonction de conversion :

Une fonction de conversion est une méthode qui effectue une conversion vers un type *T*. Elle est nommé *operator T()*. Cette méthode n'a pas de type de retour (comme un constructeur), mais doit cependant bien retourner une valeur du type *T*.

```
class Article {
  public :
```

```
    Article(double prix=0.0):_prix(prix) {}
    operator double() const { return _prix; }
  private :
    double _prix;
};

void main() {
  double total;
  Article biere(17.50);

  // utilisation implicite de la conversion Article -> double
  total = 7 * biere;
}
```
