UE D – Introduction au C++

Surcharge des opérateurs

1. Introduction

1.1. Exemple introductif

Imaginons que vous ayez défini une classe **`Complexe`** pour additionner des nombres complexes. Supposons donc que vous ayez défini quelques nombres complexes : `z1`, `z2`, etc.

A présent, on souhaiterait mettre dans `z3` le résultat de `z1 + z2`. Naturellement, le plus simple serait d'écrire quelque chose comme : `z3 = z1 + z2`.

Si vous faites ceci avec vos classes habituelles, vous allez avoir une erreur de compilation parce que justement l'opérateur **+** n'est pas défini pour les nombres complexes, donc pour `z1`, `z2`.

Une première solution serait d'écrire une fonction **`add`** qui fait l'addition entre deux nombres complexes, comme ceci : `z3 = add (z1, z2)`. Et si on voulait faire par exemple `z4` qui est le résultat de l'addition de `z1, z2, z3`, on serait obligé d'écrire `z4 = add(add(z1,z2), z3)`. Cette façon d'écrire est assez peu naturelle et qu'il serait quand même plus naturel de pouvoir écrire `z4 = z1 + z2 + z3`.

Le but de la **surcharge des opérateurs** c'est de permettre ce type d'écriture et de pouvoir redéfinir la fonctionnalité d'un opérateur + pour les objets d'une classe.

De même on pourrait imaginer afficher des nombres complexes de façon homogène, comme ceci,

Comme précédemment, cette écriture n'est pas possible si l'on ne surcharge pas l'opérateur d'affichage. Cette solution est bien préférable que d'avoir à découper ici un affichage du message " `z3 =` " puis d'appeler une fonction membre "affiche" de la classe complexe puis ensuite d'afficher le retour à la ligne.

```
cout << "z3 = " ;
Z3.afficher();
cout << endl;
```

A l'intérieur d'une classe, on peut définir des opérateurs surchargés qui ne seront applicables qu'aux objets de cette classe. Il faut néanmoins tenir compte des restrictions suivantes :

- seuls les opérateurs correspondant à des symboles définis en C++ peuvent être surchargés ;
- l'associativité (ordre d'évaluation) et la syntaxe des opérateurs ne peuvent pas être modifiés ;
- l'action des opérateurs sur les types de base ne peut pas être altérée ;
- certains opérateurs ne peuvent pas être surchargés : `., .*, ::, ?:, sizeof`

Les opérateurs C++ que l'on surcharge habituellement sont :

- Affectation, affectation avec opération `(=, +=, *=, etc)`
- Opérateur « crochets » `[]`
- Incrémentation `++`, décrémentation `--`
- Opérateur « flèche » `->`
- Opérateurs `new` et `delete`
- Opérateurs de lecture et écriture sur flux `<<` et `>>`
- Opérateurs arithmétique `(+,-,/ etc)`
- `etc`

1.2. Résolution des expressions contenant des opérateurs

La surcharge des opérateurs est possible en C++, parce qu'une expression avec un opérateur est fondamentalement résolue comme s'il s'agissait d'un appel de fonction :

```
a <Op> b
Devient : operatorOp(a, b) ou a.operatorOp(b)
```

```
<Op>  <a>
devient : operatorOp(a) ou a.operatorOp()
```

Une expression qui contient un opérateur fait appel à une fonction ou à une méthode. Écrire `a Op b` revient à appeler soit

- la fonction `operatorOp(a,b)` ;
- soit la méthode `a.operatorOp(b)` qui appartient à la classe dont **`a`** est une instance.

C'est le cas, par exemple, pour `cout << a`, qui appelle `operator<<(cout,a)` ou la méthode `cout.operator<<(a)` de la classe **`ostream`** à laquelle appartient **`cout`**. La figure ci-dessous liste les appels possibles de quelques opérateurs.

```
a + b       correspond à     operator+(a,b) ou      a.operator+(b)
b + a       correspond à     operator+(b,a) ou      b.operator+(a)
-a          correspond à     operator-(a)   ou      a.operator-()
cout << a   correspond à     operator<<(cout,a)
a = b       correspond à           -                a.operator=(b)
a += b      correspond à     operator+=(a,b)   ou   a.operator+=(b)
++a         correspond à     operator++(a)     ou   a.opertor++()
```

Note 1 : L'opérateur `operator=` correspond systématiquement à une méthode de classe et jamais à une fonction.

Note 2 : Pour des opérateurs unaires, c'est-à-dire qui ne sont liés qu'à un seul opérande, comme `-a` pour l'inverse par exemple, on appellera la fonction `operator-` avec `a` comme argument, ou la méthode `a.opertor-` sans argument.

Un opérateur peut être surchargé de deux manières différentes :

- **La surcharge externe** utilise une fonction qui prend les opérandes comme argument, prototypée et définie à l'extérieur de la classe.
- **La surcharge interne** est une méthode de la classe, dont l'instance courante est un opérande.

2. Surcharge externe

La surcharge externe consiste à définir les opérateurs comme des fonctions externes à la classe sur laquelle on travaille. La surcharge des opérateurs externes se fait donc exactement comme on surcharge les fonctions normales. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres : il n'y aura pas de paramètre implicite (le pointeur *this* n'est pas passé en paramètre).

2.1. Exemple de la classe Complexe

Supposons l'existence d'une classe **`Complexe`** et la déclaration de 3 instances, `z1, z2` et `z3` de cette classe. On souhaite faire l'addition `z3 = z1 + z2.`

Dans le cas de **surcharge externe**, c'est-à dire où l'opérateur est une fonction, cela correspond à l'appel de la fonction `z3 = operator+(z1,z2)`. Son prototype va prendre deux arguments de type Complexe et va retourner un Complexe.

Pour optimiser cet appel, il est possible de faire **passer les arguments par référence constante** par **le biais du `const&`**, et également de qualifier le type de retour par `const`. Les différents prototypes possibles de cette fonction sont listés ci-dessous :

```
// Prototype de base :
Complexe operator+(Complexe z1, Complexe z2);

// Prototype optimisé:
const Complexe operator+(Complexe const& z1, Complexe const& z2);
```

Notez que le type de retour de cet opérateur est un « `const`. Cela permet d'éviter, par exemple d'incrémenter l'addition de « `z1,z2` » ; cela évite ce genre d'expression :

- `++ (z1 + z2) ;`

Ainsi, on ne pourra pas modifier le résultat d'une addition entre deux complexes qui est une constante (**voir la section 2.3 du polycopie bases du C++**).

2.2. Exemple où la surcharge externe est nécessaire

Il existe des situations où la surcharge externe est nécessaire : c'est le cas quand les opérandes sont de type différent ou quand on veut surcharger l'opérateur d'affichage.

Considérons le cas de la multiplication d'un nombre Complexe avec un double. Par exemple, on a ici un double " `a` " et un nombre Complexe " `z1` ", et on veut récupérer le résultat de `a * z1` dans `z2`.

Donc, `z2 = a * z1` nous donnerait comme réécriture soit :

- `z2 = a.operator*(),` avec une surcharge interne et appel d'une méthode,
- `z2 = operator*(a,z1),` avec une surcharge externe et appel d'une fonction.

La première option n'est pas correcte puisque `a` n'est pas une instance de classe, mais un type de base, de type `double`. Là on est obligé d'utiliser la surcharge externe. L'appel équivalent serait `z2 = operator*(a,z1),` ce qui nous donne donc le prototype suivant de la fonction qui retourne un nombre Complexe.

```
double a;
Complexe z1, z2;
// . . .

Z2 = a * z1;

const Complexe Complexe::operator*(double x, Complexe const &z)
{
   return  z * x;
}
```

Pour la définition de la fonction, on pourrait écrire explicitement toutes les opérations qui font la multiplication du double, `a`, par le Complexe `z`.

► Dans la définition fonction `a.operator*()`, il est tout à fait possible de faire le calcul avec l'écriture `z * a` (voire la définition ci-dessus). Ce calcul peut alors correspondre à une méthode `z.operator*(a)` au sein de la classe Complexe. L'intuition ici est d'éviter de réécrire 2 fois le même code car les deux expressions (`a * z`) et (`z * a`) sont équivalentes.

a) Surcharge de l'opérateur d'affichage cout de la classe Ostream

De même, dans le cas où on voudrait afficher un complexe, par exemple faire " `cout << z1` " en utilisant la surcharge de l'opérateur d'affichage `<<`. On aurait donc deux écritures possibles :

- soit la méthode "`operator<<`" de la classe `ostream` dont `cout` est une instance, avec un appel de la méthode de cette classe, `cout.operator<<(a)`
- soit la fonction qui prend ici les 2 paramètres `cout` et `z1` : `operator<<(cout,a)`

Mais ce qui nous intéresse est de surcharger la classe Complexe et non pas de toucher à la classe `ostream` dont `cout` est une instance, et donc dans ce cas aussi, on ne va pas utiliser la surcharge interne à la classe `ostream`, mais bien préférer la surcharge externe. Le prototype de cette fonction est le suivant :

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

Il est important de passer le premier argument par référence puisque l'affichage va modifier le flot en question. Les flots ne peuvent d'ailleurs être passés que par référence.

Notons aussi que la valeur de retour de l'opérateur est une référence sur une instance de la classe `ostream`.

L'expression `cout << z1 << endl` est équivalente à `operator<<(cout << z1, endl) ;` qui est elle-même équivalente à `operator<<(operator<<(cout,z1),endl).`

Donc la valeur de retour de `operator<<(cout,z1)` est passé comme premier argument d'un autre appel à `operator<<.` Cette expression doit être de même type que le premier argument de l'opérateur `".` Cette expression doit être également de type `ostream&` pour pouvoir être modifié à son tour.

Si vous écrivez void `operator<<(...);` c'est à dire que la valeur de retour est de type void, dans ce cas le second `<<` de l'expression `cout << z1 << endl` ne pourrait pas avoir lieu puisque la première expression `cout << z1` va retourner un void, ce qui n'a pas de sens.

Pour définir le corps de la fonction, on peut imaginer plusieurs alternatives, décrites ci-dessous :

- La première consisterait à utiliser des accesseurs qui retournent les attributs que l'on souhaite afficher ;
- Une deuxième alternative peut utiliser une méthode préalablement définie, qui, par exemple, affiche un complexe, comme affiche() qui prend en paramètre le flot à modifier et réalise l'affichage demandé.

```
// Via des accesseurs:
ostream& operator<<(ostream& sortie, Complexe const& z) {
   sortie << "(" << z.getX() << "," << z.getY() << ")";
   return sortie;
}
```

```
// Via une méthode:
ostream& operator<<(ostream& sortie, Complexe const& z) {
   sortie << "(" << z.getX() << "," << z.getY() << ")";
   return z.affiche(sortie);
}
```

b) Accès aux membres privés

Lors de surcharges externes, il est parfois nécessaire d'accéder aux attributs privés de la classe sur laquelle on veut faire porter l'opérateur, c'est à dire aux données membres de leurs opérandes. Il est alors fortement recommandé de toujours utiliser l'interface publique de la classe, les accesseurs.

Il est aussi possible de donner un accès privilégié à la fonction en la déclarant en tant que *fonction amie*, avec le mot-clé friend. Cette « friendship » entre la fonction externe et la classe lui confère le droit d'accéder à ses éléments privés.

Les opérateurs externes doivent être déclarés comme étant des fonctions amies de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

Dans notre exemple précédent, si l'on déclare un tel lien entre la fonction de surcharge de l'opérateur d'affichage et la classe Complexe, on peut ainsi utiliser directement les attributs privés des instances, mais, à nouveau, il est peu conseillé d'employer cette technique.

```
class Complexe {

   private :
   double x;
   double y;

   friend ostream& operator<<(ostream& sortie, Complexe const& z);
... } ;
```

c) Exemple complet

Considérant une classe **Entier** contenant un élément de type int. On souhaite redéfinir l'opérateur d'addition permettant d'additionner deux instantes de la classe **Entier**, ou encore d'addition une constante avec une instance de la classe **Entier**.

L'exemple ci-dessous montre quelques exemples de surcharges de l'opérateur + en tant que fonction amie de la classe **Entier**.

```
#include <iostream>

using namespace std;

class Entier {
   private :
   int i;

   public:
       Entier(int x = 0): i(x) {}
       friend Entier operator+(Entier const &p1, Entier const &p2);// I
       friend Entier operator+(int x, Entier const &p);           // II
       friend Entier operator+(Entier const &p1, int x);          // III
} ;

Entier operator+(Entier const &p1, Entier const &p2){

       return Entier(p1.i+p2.i);
}

Entier operator+(int x, Entier const &p){

       return Entier(x+p.i);
}

Entier operator+(Entier const &p, int x){

       return Entier(p.i+x);
}
```

```
int main(int argc, char **argv)
{

  Entier a1, a2(2), a3(3);
      a1 = a2 + a3; // Appelle la version I
  a1 = a2 + 10; // Appelle la version II
      a1 = 10 + a2; // Appelle la version III
  return 0;
}
```

3. Surcharge interne

Un opérateur peut également être surchargé à l'intérieur d'une classe : dans ce cas-là, il s'agira d'une méthode de la classe. Dans ce cas, l'opérateur s'appliquera à ses instances. Le premier opérande est alors l'instance courante sur laquelle on appelle la méthode, et le deuxième (éventuel) sera son unique argument, puisqu'elle a déjà accès aux attributs de l'instance courante.

3.1. Surcharge interne des opérateurs

Pour surcharger un opérateur `Op` dans une classe `NomClasse`, il faut ajouter la méthode `operatorOp` dans la classe en question :

```
class NomClasse {
...
// prototype de l'opérateur Op
type_retour operatorOp(type_parametre);
...
};
// définition de l'opérateur Op
type_retour NomClasse::operatorOp(type_parametre)
{
...
}
```

Les opérateurs définis en interne devront souvent renvoyer l'objet sur lequel ils travaillent (ce n'est pas une nécessité cependant). Cela est faisable grâce au pointeur *this*.

3.2. Exemple de surcharge interne des opérateurs

On s'intéresse ici à la surcharge de l'opérateur `+=` en interne dans la classe « Complexe ». Cet opérateur a la sémantique suivante : ajouter à `z1` la valeur de `z2` ».

Contrairement à « l'opérateur + », qui va construire un nouveau Complexe à partir de deux Complexes existants, l'opérateur `+=`, lui, va modifier son premier opérande.

L'opérateur `+=` est donc un opérateur « proche de la classe » puisqu'il permet d'en modifier le contenu, et dans ce cas on a recours à une une surcharge interne, ce qui va se traduire par le prototype et la définition suivante :

```
class Complexe {
   public:
   // ...
   void operator+=(Complexe const& z2); // z1+=z2
   // ...
   };

void Complexe::operator+=(Complexe const& z2) {
x += z2.x;
y += z2.y;
}
```

Il est tout à fait possible d'établir un lien sémantique entre les deux les opérateurs `« += »` et `'+'` en exprimant l'un en fonction de l'autre.

- Si l'on utilise la définition ci-dessous de l'opérateur '+', lorsque l'on écrit `c1 + c2`, la valeur de `c1` va être copiée dans `z1` qui est une copie locale à opérateur `'+'`. Cette variable temporaire sera la valeur de retour.

```
class Complexe {
   public:
   // ...
   void operator+=(Complexe const& z2); // z1+=z2
   const Complexe operator+(Complexe z1, Complexe const& z2);
   };

const Complexe Complexe::operator+(Complexe z1, Complexe const& z2) {
z1 += z2; // utilise l'opérateur += redéfini précédemment
return z1;
}
```

a) Quel type de retour pour les opérateurs d'auto-affectation ?

Dans notre exemple précédent, le type de retour de l'opérateur `+=` est de type `void.`

Toutefois, on pourrait bien écrire `z3=(z1+=z2)`. Le résultat de `z1+=z2`, qui sera la valeur de `z1` après l'opération d'auto-affectation, pourra être stocké dans `z3.`

Il s'agit donc d'une instance de même classe que l'opérande `z1`, avec la même valeur. Le type de retour est donc un Complexe et non pas void. Toutefois, comme `z1` existe déjà, on va pas recréer un nouveau complexe mais on va passer une référence vers un complexe qui existe déjà.

Pour éviter une copie inutile, on peut donc retourner une référence sur `z1`.

Ainsi, le type de retour des opérateurs d'auto-affectation est une référence sur l'instance courante, de la forme Classe& . On mettra une référence à chaque fois que l'on retourne un complexe qui existe déjà. Ce n'est pas une nouvelle valeur. Dans cette expression ici, on retourne la valeur de `z1`.

```
class Complexe {
public:
// ...
Complexe &operator+=(Complexe const& z2); // z1+=z2
// ...
};

Complexe& Complexe::operator+=(Complexe const& z2) {
x += z2.x;
y += z2.y;

return *this;
}
```

Notons que l'opérateur retourne l'instance courante, c'est à dire la valeur qui est stockée à l'adresse de l'instance courante, donc `*this,` c'est-à-dire par exemple ici, la valeur de `z1`.

Les opérateurs d'affectation fournissent un exemple d'utilisation du pointeur *this*. Ces opérateurs renvoient en effet systématiquement l'objet sur lequel ils travaillent, afin de permettre des affectations multiples. Les opérateurs de ce type devront donc tous se terminer par : return *this;

b) Éviter les copies

Dans l'exemple précédent, le type de retour est une référence puisque l'instance résultante existe déjà ceci permet d'éviter des copies inutiles. De manière générale, il est important d'optimiser la surcharge d'opérateurs à l'aide de références afin de limiter toute copie superflue.

Considérons la variante suivante de surcharge de l'opérateur +=.

```
class Complexe {
public:
// ...
Complexe operator+=(Complexe z2); // z1+=z2
// ...
};

Complexe Complexe::operator+=(Complexe z2) {

Complexe z3;
x += z2.x;
y += z2.y;
z3 = *this,
return z3;
}
```

Dans cette variante, tout d'abord, on effectue un passage par valeur de l'argument z2 et donc possiblement une première copie.

Concernant la valeur de retour, si nous avons une expression `z3=(z1+=z2),` nous aurons par valeur de retour une nouvelle instance de la classe, qui sera copiée puisqu'elle n'est pas passée par référence, contrairement à la première version. Enfin, dans le corps de la méthode, cette version déclare la nouvelle instance Complexe qui sera retournée, dans laquelle on copie l'instance courante modifiée (z3 = *this).

c) Surcharge interne ou externe ?

- Préférer la **surcharge externe** lorsque le corps de la fonction peut être écrit sans avoir recours au mot-clé friend qui brise l'encapsulation et sans devoir coder des accesseurs. Dans ce cas, l'opérateur doit pouvoir être défini par le biais de méthodes publiques de la classe : un exemple est l'opérateur + vu précédemment, qui exploite le += .

- Si l'opérateur doit modifier une instance : pour lui donner un accès interne, on préférera alors une surcharge interne.

4. Compléments sur la surcharge d'opérateurs

4.1. Opérateur d'affectation

L'affectation est l'une des opérations les plus fondamentales de tout langage. Cet opérateur, est utilisé par exemple chaque fois qu'on écrit `« a = b »`, `« z1 = z2 »`.

L'appel équivalent de cet opérateur d'affectation est `« a.operator = »`, on a recours à une une surcharge interne, ce qui va se traduire par le prototype et la définition suivante. Pour éviter une copie, on va le passer par référence constante.

```
class NomClasse {
...
// prototype de l'opérateur =
NomClasse & operator=(NomClasse source);
...
};

// définition de l'opérateur =
NomClasse & NomClasse::operatorOp(NomClasse source)
{
// faire la copie entre les données membres

return *this ;

}
```

Cet opérateur renvoie une référence sur `NomClasse` afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite : a=b=c est évaluée comme a=(b=c). Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

Notons que l'implémentation des opérateurs d'affectation peut parfois soulever quelques problèmes.

- Par exemple, si vous ne redéfinissez pas le constructeur de copie, les écritures telles que : classe object = source; ne fonctionneront pas correctement. En effet, c'est le constructeur de copie qui est appelé ici, et non l'opérateur d'affectation comme on pourrait le penser à première vue.
- Lorsque l'on écrit un opérateur d'affectation, on a généralement à reproduire, à peu de choses près, le même code que celui qui se trouve dans le constructeur de copie.
- Un autre problème important est celui de l'autoaffectation. Non seulement affecter un objet à lui-même est inutile et consommateur de ressources, mais en plus cela peut être dangereux. Une solution simple consiste ici à ajouter un test sur l'objet source en début d'opérateur, comme dans l'exemple suivant :

```
classe &classe::operator=(const classe &source)
{
        if (&source != this)
                {
                        // Traitement de copie des données :
                        ...
                }
        return *this;
}
```

4.2. Exemple de surcharge avec l'opérateur =

```
#include <iostream>

using namespace std;

class Entier {
   private :
   int i;

   public:
       Entier& operator=(Entier &p)
   {
     i = p.i;
     return *this;
   }
} ;

int main(int argc, char **argv)
{

   Entier a1, a2;
      a1 = a1 + a2; // Appel de l'affectation surchargée
   return 0;
}
```