Surcharge d’opérateurs

La surcharge d’un opérateur est l’opération qui consiste à donner un sens à un opérateur appliqué à un type nouveau, pour lequel il n’a pas été prévu dans le langage c++.

En cohérence avec le modèle objet, le c++ permet de (re)définir le comportement des opérateurs pour les appliquer à des objets. En effet, il peut être utile/pratique/simple/agréable et surtout avoir du sens d’écrire quelque chose comme :

```
Class_Truc A, B, C ;

…

C = A + B ;
std::cout << C ;
…
```

Mais bien sûr les opérateurs + ou << n’étant pas définis sur la classe Class_Truc, ces expressions généreront des erreurs à la compilation.

Mécanisme

La façon de faire proposée par le c++ est très simple, cela consiste à dire que derrière chaque opérateur il y a une fonction et comme il est toujours possible de redéfinir une fonction à condition de changer au moins un des types de ses arguments, il est possible de redéfinir un opérateur !

Syntaxiquement la fonction associée à l'opérateur **@** est de la forme : `operator@(…)`, par exemple la fonction associée à l'opérateur **+** se nomme `operator+(…)`.

Les opérandes de l'opérateur sont les arguments de la fonction.
Dans le cas d'une fonction membre l'argument implicite constitue le premier opérande de l'opérateur.

Contraintes

- on peut redéfinir un opérateur, mais on ne peut pas changer ses caractéristiques, c'est à dire qu'on ne peut pas changer sa priorité, son sens d'associativité ni son arité (nombre d'opérandes),
- comme ce mécanisme est prévu pour donner du sens à des opérateurs appliqués à des types nouveaux, au moins un des opérandes doit être de type classe,
- les opérateurs commutatifs perdent cette propriété,
- tous les opérateurs du C++ peuvent être surchargés sauf les 5 :

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

Exemple

On définit une classe complexe pour gérer des nombres complexes. Il est intéressant de disposer de certains opérateurs pour une écriture plus simple des programmes utilisant des complexes :

on peut vouloir écrire des expressions utilisant l'addition de complexes et dans ce cas une écriture avec l'opérateur **+** est certainement plus lisible et pratique qu'une écriture utilisant des fonctions (`Somme` ou `Add` par exemple).

Il faut donc définir la fonction `operator+(…)` appliquée à 2 complexes pour pouvoir utiliser l'opérateur. 2 possibilités : fonction membre ou fonction non membre. Pour écrire une fonction membre il faut :
- que le premier argument soit un objet de la classe,
- avoir accès (être le programmeur) de la classe.

l'idée est que A + B correspond en fait à l'appel de :
- soit, dans le cas d'une fonction membre : `A.operator+(B)`
- soit, dans le cas d'une fonction non membre : `operator+(A, B)`

On va définir dans le 1er cas :

```
complexe complexe::operator+(complexe X)
{
 complexe R ;
 R.r = (*this).r + X.r ;
 R.i = (*this).i + X.i ;
 return R ;
}
```

et on va définir dans le 2nd cas :

```
complexe operator+(complexe X, complexe Y)
{
 complexe R ;
 R.r = X.r + Y.r ;
 R.i = X.i + Y.i ;
 return R ;
}
```

Un autre exemple, on veut pouvoir comparer les valeurs de 2 complexes avec l'opérateur ==, pour cela on pourra écrire :

```
bool complexe::operator==(complexe C)
{
 return (*this).r == C.r && (*this).i == C.i;
}
```

ou

```
bool operator==(complexe X, complexe Y)
{
 return X.r == Y.r && X.i == Y.i;
}
```

Cas particulier l'opérateur d'affectation =
L'opération d'affectation que l'on peut résumer en "donner une valeur à une variable" se réalise par l'opératuer = :

```
i = 3
cpt = 0
cpt = cpt + 1
pair = true
```

...

Dans le cas d'un variable objet d'une classe qu'on a créée, le compilateur fabrique un operator= simple qui consiste juste à affecter tous les membres de l'objet.

```
Point p1(2,3), p2;
p2 = p1;  // Pas de problèmes
```

Si les objets concernés contiennent un membre dynamique, celui-ci ne sera pas dupliqué, mais le pointeur sera juste copié et les 2 objets pointeront vers la même zone allouées.

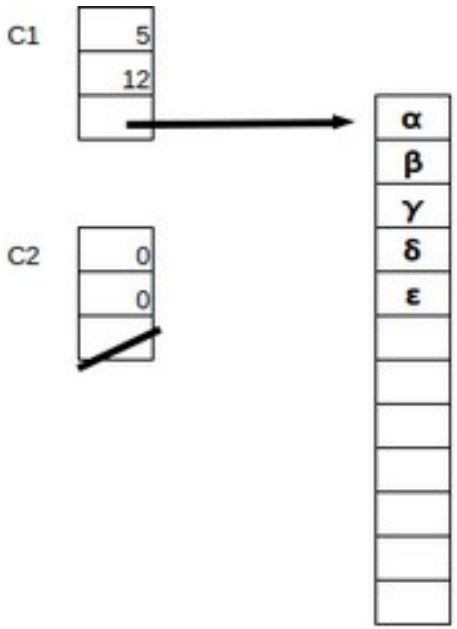
Il convient donc de créer la fonction operator= qui prendra en charge la duplication des zones dynamiques, avec les allocations nécessaires et les libérations éventuelles.

Enfin cette fonction retournera l'objet lui-même pour respecter le fonctionnement de l'opérateur = et pour permettre des écritures du type :

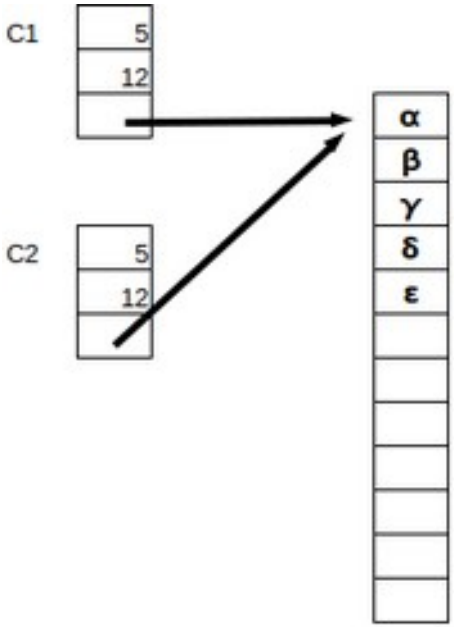
```
a = b = c = d;
```

```
col_Points C1(12), C2;
```

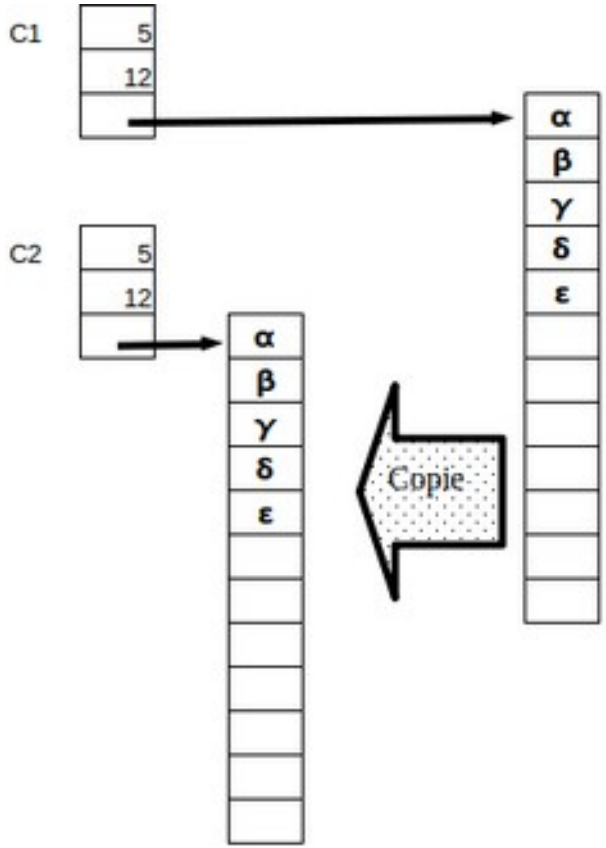
// Remplissage de C1 et avec les 5 points **α**, **β**, **γ**, **δ**, **ε**




```
C2 = C1;
```



avec un = bien défini :




```
col_Points::operator=(col_Points & C)
{
  if (&p != this)
  {
    if (tab != NULL)
      delete [] tab;
    nbp = C.nbp;
    cap = C.cap;
    tab = new Point[cap];
    for (int i = 0;i < nbp; i++)
      tab[i] = C.tab[i];
  }
  return *this;
}
```

Dernier exemple, on veut afficher un complexe en utilisant l'opérateur << . Il faut donc définir la fonction `operator<<` de manière à pouvoir écrire :

```
complexe C ;
…
cout<< C ;
```

Dans ce cas, l'appel correspond à :
- fonction membre : **`cout.operator<<(C)`**
- fonction non membre : **`operator<<(cout, C)`**

Le premier argument étant `cout`, si on veut écrire une fonction membre, celle-ci sera membre de la classe de cout. Or nous n'avons pas accès à cette classe, nous ne pouvons pas y ajouter de fonction. Donc la seule solution est d'écrire une fonction non membre.
Cette fonction doit respecter les contraintes de l'opérateur (voir plus haut), elle doit donc pouvoir modifier le premier opérande et elle doit le renvoyer pour pouvoir écrire des expressions comportant plusieurs fois l'opérateur :

```
ostream & operator<<(ostream & OUT, complexe C)
{
 OUT << C.r << " + i*" << C.i;
 return OUT;
}
```

Dernier point, cette fonction n'étant pas une fonction membre, elle n'a pas le droit d'accéder aux données membres des objets de la classe complexe. Une façon simple et propre de régler ce problème est de le déclarer « amie » dans la classe complexe en ajoutant son en-tête préfixé par « friend » dans la classe.

```
 friend ostream & operator<<(ostream & OUT, complexe C);
```

Au final voici un petit programme minimal qui reprend l'exemple :

```
#include <iostream>

using namespace std;
class complexe
{
 private :
     double r, i ;

 public :
     complexe(double rr = 0, double ii = 0) {r = rr; i = ii;}
     complexe operator+(complexe C)
     {
         complexe R((*this).r + C.r, (*this).i + C.i);
         return R;
     }

     bool operator==(complexe C)
     {
         return (*this).r == C.r && (*this).i == C.i;
     }
     friend ostream & operator<<(ostream & Sout, complexe C);
};

ostream & operator<<(ostream & OUT, complexe C)
{
 OUT << C.r << " + i*" << C.i;
 return OUT;
}
```

```
int main()
{
 complexe A, B(2,3), C(5,4);

 A = B + C;

 cout << B << " + " << C << " = " << A << endl;

 complexe D(7, 7), E;

 if (D == A)
     cout << D << " == " << A << endl;
 else
     cout << D << " != " << A << endl;

 if (E == A)
     cout << E << " == " << A << endl;
 else
     cout << E << " != " << A << endl;
}
```

On obtient à l'exécution :

```
2 + i*3 + 5 + i*4 = 7 + i*7
7 + i*7 == 7 + i*7
0 + i*0 != 7 + i*7
```

Cas particulier des opérateurs ++ et - -

Ces 2 opérateurs existent sous 2 versions : **pré**(inc, dec)rémentation et **post**(inc, dec)rémentation.

La version pré est surchargée en utilisant la fonction `operator` normale et la version post est surchargée par la fonction `operator` à laquelle on ajoute, lors de sa déclaration, un argument de type `int` factice.

Exemple sur le type point où l'incrémentation d'un point correspondrait à une incrémentation des 2 coordonnées.

Version fonctions membres :

```
Class Point
{
 int x, y ;
 public :
 …
 // preincrementation
 point operator++()
 {
     (*this).x ++ ;
     (*this).y ++ ;
     return (*this) ;
 }

 // postincrementation
 point operator++(int)
         // argument int factice
 {
         // sauvegarde de la valeur
         // de l'objet pour la retourner
     Point tmp = *this ;
         // utilisation de la surcharge
         // de la preincrementation
     ++(*this) ;
         // retour de la valeur
         // avant incrementation
     return tmp;
 }
} ;
```

Version fonction non membres :

```
// preincrementation
point operator++(point & P)
{
    P.x ++ ;
    P.y ++ ;
    return (P) ;
}

// postincrementation
point operator++(point & P, int) // argument int factice
{
    // sauvegarde de la valeur de l'objet pour la retourner
    Point tmp = P ;
    // utilisation de la surcharge de la preincrementation
    ++(P) ;
    // retour de la valeur avant incrementation
    return tmp;
}
```

Avec bien sûr ajout dans la classe point :

```
friend complexe operator++(complexe &);
friend complexe operator++(complexe &, int);
```