Copie et affectation

1. Constructeurs
2. Constructeur de copie
3. Affectation
4. Conversions

La copie et l'affectation sont les opération très importante. Elles sont à la base des effets de bord et de la gestion de la mémoire.

Constructeurs

Les constructeurs se classent syntaxiquement en 4 catégories

1. le constructeur par défaut, sans argument,
2. les constructeurs de conversion, à un argument
3. le constructeur de copie, à un argument
4. les autres constructeurs.

Un constructeur de conversion sert souvent à la promotion.

Exemple:

```
class Rat {
  int num, den;
public:
  int num, den;
  ...
  Rat(int n)  { num = n; den = 1;}
  ...
};
```

On s'en sert pour la conversion

```
Rat r, s;
...
r = s + 7;
```

L'entier `7` est promu en `Rat(7)` avant l'addition.

Constructeur de copie

Chaque classe possède un constructeur de copie (ne pas confondre avec l'opérateur d'affectation).

Le constructeur de copie par défaut copie membre à membre en utilisant le constructeur de copie de chaque membre, et une copie bit à bit pour les types de base (copie superficielle). Ceci est insuffisant en présence d'adresses. Il faut alors allouer de nouvelles zones mémoire (copie profonde).

Le constructeur de copie est utilisé

* lors d'une initialisation:

```
X  x = y;
X  x(y);
X* x = new X(y);
```

où y est un objet déjà existant.

* lors du retour d'une fonction retournant un objet par valeur;
* lors du passage d'un objet par valeur à une fonction.

Le constructeur de copie a deux prototypes possibles :

```
X (X&);
X (const X&);
```

L'argument est donc un objet de la classe X passé par référence (évidemment !) et éventuellement spécifié comme non modifiable dans le constructeur.

Voici une classe Pt possédant:

* Un constructeur de copie `Pt(const Pt& p)`
* Un opérateur d'affectation `Pt& operator=(const Pt& p)`

```
#include <iostream.h>
class Pt {
  int x, y;
public:
  Pt(int abs = 1, int ord = 0) x(abs), y(ord) {}

  //implantation par défaut:
  Pt(const Pt& p) : x(p.x), y(p.y){}

  //implantation par défaut:
  Pt& operator=(const Pt& p) {
    x = p.x; y = p.y;
    return *this;
  }
};
```

Des fonction auxiliaires

```
void Fn(Pt p, Pt& q, Pt* r) {
  delete r;
}

Pt vv(Pt x) { return x; }
Pt vr(Pt& x) { return x; }
Pt& rv(Pt x) { return x; }
Pt& rr(Pt& x) { return x; }
```

```
void Fn(Pt p, Pt& q, Pt* r) {
  delete r;
}

Pt vv(Pt x) { return x; }
Pt vr(Pt& x) { return x; }
Pt& rv(Pt x) { return x; }
Pt& rr(Pt& x) { return x; }
```

```
void main()
{
 Pt a;          //Construction de a : 1 0
 Pt b = 3;      //Construction de b : 3 0
 Pt c = Pt(0,2); //Construction de c : 0 2
 {   Pt d = Pt(a); //Recopie     de a en d
     Pt e = a;     //Recopie     de a en e
                   //Destruction  de e
 }                 //Destruction  de d
 Pt f;          //Construction de f : 1 0
 Pt* p = new Pt(6,7);//Construction de *p : 6,7
 c = f;             //Affectation  de f a c
 c = Pt(5,5);       //Construction de temp : 5 5
                    //Affectation  de temp a c
                    //Destruction  de temp

 Fn(a, b, p);       //Recopie     de a en abis
                    //Destruction  de *p
                    //Destruction  de abis

 f = vv(b);  //Recopie     de b en bis
             //Recopie     de bis en bpile valeur de retour
             //Destruction  de bis
             //Affectation  de bpile a f
             //Destruction  de bpile
 f = vr(b);      //Recopie     de b en bpile
             //Affectation  de bpile a f
             //Destruction  de bpile
 f = rv(b);      //Recopie     de b en bis
             //Destruction  de bis
             //Affectation  de bis a f // danger !
 f = rr(b);      //Affectation  de b a f(sans copie)
}
```

Affectation

L'affectation et la copie sont des opérations différentes.

* le constructeur de copie sert à l'initialisation et pour le passage de paramètres. Il construit l'objet.
* l'opérateur d'affectation = pour l'affectation dans une expression. Il retourne un objet ou une référence.

L'opérateur d'affectation est un opérateur, et peut donc être redéfini explicitement par une fonction operator=().

Chaque classe possède un opérateur d'affectation par défaut.

L'affectation est superficielle comme la copie, et consiste en une affectation membre à membre.

La signature de l'opérateur est

```
X& operator=(X&)
X& operator=(const X&)
X operator=(X&)
X operator=(const X&)
```

et l'expression y = x équivaut à

```
y.operator=(x)
```

Le résultat de l'affectation est donc dans l'objet appelant. A quoi équivaut x = y = z = t ? Voici une petite classe Id:

```
class Id {
  char* nom;
public:
  Id(const Id&);
  Id& operator=(const Id&);
};
```

Le constructeur de copie est:

```
Id::Id(const Id& x) {nom = new char[1+strlen(x.nom)];strcpy(nom,x.nom);}
```

L'opérateur d'affectation est:

```
Id& Id::operator=(Id& x){
  if (this == &x) // même objet ne rien faire
    return *this;
  if (x.nom == 0)
    nom = 0;
  else {
    delete nom;
    nom = new char[1+strlen(x.nom)];
    strcpy(nom,x.nom);
  }
  return *this;
}
```

La référence retournée est celle de l'objet appelant.

Chaque méthode contient un pointeur sur l'objet appelant la méthode; ce pointeur a pour nom `this`.
L'objet lui-même est donc `*this`.

Conversions

Les conversions définies par le programmeur sont
* les constructeurs de conversion;
* les opérateurs de conversion.

Un opérateur de conversion d'une classe C en un classe ou un type T

* est une fonction membre de C;
* qui a le prototype `operator T()`. Le type de retour est donc `T`
* est appelé implicitement pas l'objet,
* n'est utilisé que si nécessaire.

Exemple. On veut pouvoir écrire

```
Rat r;
...
if (r) ...
```

C'est possible en convertissant r en `bool` qui est vrai si le numérateur de r est non nul.

```
class Rat{
...
  operator bool() { return (num) ? 1 : 0;}
}
```