

## Chapitre 7

### Construction, destruction, initialisation et recopie

### 1. Les 4 outils de départ

Pour une classe quelconque, le C++ fournit par défaut pour une classe donnée:

- un constructeur sans argument (n'initialise rien)
- un constructeur de recopie
- un destructeur
- un opérateur d'affectation

2. Durée de vie d'un objet



\* création

- déclaration (objets statiques ou automatiques)
- new (objets dynamiques)

\* mort

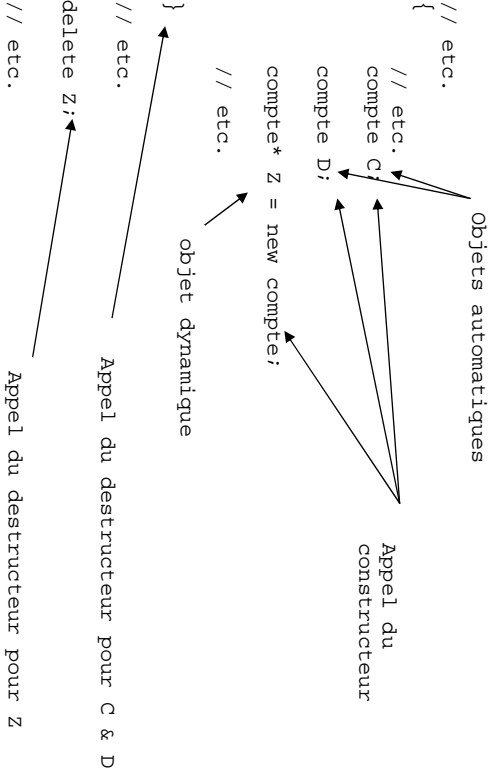
- fin de la portée (objets statiques ou automatiques)
- delete (objets dynamiques)

3. Construction et destruction d'un objet

\* constructeur: appel automatique juste après la création de l'objet

\* destructeur: appel automatique juste avant la mort de l'objet

Le constructeur et le destructeur assurent que l'objet est dans un état cohérent.

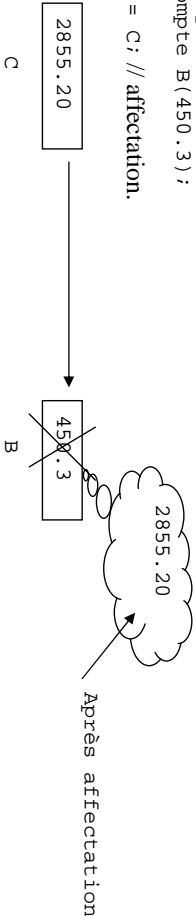


4. Affectation

Par défaut, affectation membre par membre.

```
class compte {  
    double actif;  
public:  
    // etc.  
};
```

```
compte C(2855,20);  
compte B(450,3);  
B = C; // affectation.
```



Cet opérateur sera développé plus longuement dans le chapitre "surcharge des opérateurs".

5. Constructeur de recopie

De la même manière que l'affectation (par défaut), le constructeur de recopie par défaut effectue la copie membre à membre de la source vers la destination.

Il est appelé dans trois situations:

5.1. Initialisation d'un objet

Création d'un nouvel objet initialisé comme étant une copie d'un objet existant:

```
compte C(280.98);  
  
compte B(C); // constructeur de recopie: création de l'objet B et son initialisation avec les données de C.
```

Ou bien

```
compte B = C; // constructeur de recopie: création de l'objet B et son initialisation avec les données de C.
```

Attention ... différencier entre recopie et affectation:

```
compte B; // création de l'objet.  
  
B = C; // pas de recopie, mais l'affectation, car objet déjà créé.
```

## 5.2. Paramètre passé par valeur

Transmission d'une valeur à une fonction

```
compte C(290.77);

double fonction(compte);

double z = fonction(C); // passage par valeur de C.
```

## 5.3. Retour de fonction par valeur

```
compte C;

compte fonction(int);

compte fonction() {
    compte X;
    return X; // retour par valeur de X
}
```

## 5.4. Utilité d'un constructeur de recopie

```
#include <iostream>
class test {
    int *ptr; // on suppose que nous avons un pointeur sur un seul élément.
    void alloc_test() {
        if (ptr==NULL) {
            std::cerr << "allocation de la mémoire a échoué!\n";
            exit(1);
        }
    }
public:
    test(int d=1000) {
        ptr = new int(d); // allocation et initialisation.
        alloc_test();
    }
    ~test() {
        delete ptr; // libération.
    }
    void affiche() { std::cout << "valeur: " << *ptr << std::endl; }
};

int main() {
    test x; // appel du constructeur, déclaration permise, car, par défaut: d=1000.
    test y = x; // appel du constructeur de recopie dans ce cas, par défaut fourni par le langage création
                // de l'objet y puis recopie membre à membre de x vers y.
    y.affiche();
    return 0;
}
```

Le constructeur de recopie par défaut va effectuer une copie membre à membre. Copier membre à membre un pointeur signifie que `ptr` de l'objet `y` pointe au même endroit que `ptr` de l'objet `x` (une copie d'adresse).

À la fin du programme, le destructeur va être appelé pour détruire les objets `x` et `y`.

Détruire `x` revient à libérer la mémoire allouée pour le pointeur `ptr` de `x`.

Détruire `y` revient à libérer la mémoire allouée pour le pointeur `ptr` de `y`.

Or, comme mentionné précédemment, les deux pointeurs pointent le même endroit, de ce fait, le même endroit va être détruit deux fois ! Or, on ne peut pas détruire successivement deux fois la même chose, d'où, lors de l'exécution du programme, nous obtenons :

Segmentation fault (core dumped)

C'est une erreur fatale, classique, quand il y a violation de la mémoire. Une tentative d'accéder à quelque chose qui n'existe pas.

Pour corriger cette erreur, il faut définir le constructeur de recopie, afin de masquer le constructeur de recopie par défaut fourni par le langage et redéfinir le nôtre.

### 5.5. prototype du constructeur de recopie

```
nom_classe (const nom_classe&) ;
```

- porte le même nom que la classe,
- accepte comme argument un objet du type de la classe dans laquelle il a été déclaré, il est passé par référence et il est constant, car l'objet ne sert que pour la recopie.
- ne retourne rien (ne pas mettre `void`)

Pour l'exemple du paragraphe 5.4:

```
#include <iostream>

class test {

    int *ptr; // on suppose que nous avons un pointeur sur un seul élément.
    void alloc_test() {
        if (ptr==NULL) {
            std::cerr << "allocation de la mémoire a échoué!\n";
            exit(1);
        }
    }

public:
    test(int d=1000) {
        ptr = new int(d); // allocation et initialisation.
        alloc_test();
    }

    test(const test& T) {
        ptr = new int(*T.ptr);
        alloc_test();
    }

    ~test() {
        delete ptr; // libération.
    }

    void affiche() {
        std::cout << "valeur: " << *ptr << std::endl;
    }
};
```

```
int main() {
    test x; // appel du constructeur, déclaration permise car par défaut: d=1000.

    test y = x; // appel du constructeur de recopie.
    y.affiche();

    return 0;
}

Sortie:

valeur: 1000
```

Un constructeur de recopie par défaut copie donc bit à bit une zone de mémoire dans une autre, on parle alors de *copie superficielle*. Ceci est insuffisant en présence de pointeurs comme membres de données dans une classe. Il faut alors allouer une nouvelle zone mémoire, on parle alors de *copie profonde* (allocation puis recopie).

6. Objet contenant un objet

```
#include <iostream>

class compte {
    double actif;
public:
    compte(double d):actif(d){}
    void affiche(){
        std::cout << "actif: " << actif << std::endl;
    }
};

class client {
    compte c;
    int nas;
public:
    client(int m, double v):c(v),nas(m) {}
    void affiche() {
        std::cout << "nas: " << nas << std::endl;
        c.affiche();
    }
};

int main() {
    client A(231940,456.89);
    A.affiche();

    return 0;
}
```

Sortie:

```
nas: 231940
actif: 456.89
```

Pour créer un client, il faut passer par le constructeur de `client` qui, lui, doit appeler le constructeur de `compte`. En premier, ce sont les membres données de `compte` qui sont initialisés puis sera le tour des autres membres données de `client`.