Chapitre 14

Construction, destruction, initialisation et recopie

1. Les 4 outils de départ

- Pour une classe quelconque, le C++ fournit par défaut :
  - un constructeur sans argument (n'initialise rien)
  - un constructeur de recopie
  - un destructeur
  - un opérateur d'affectation

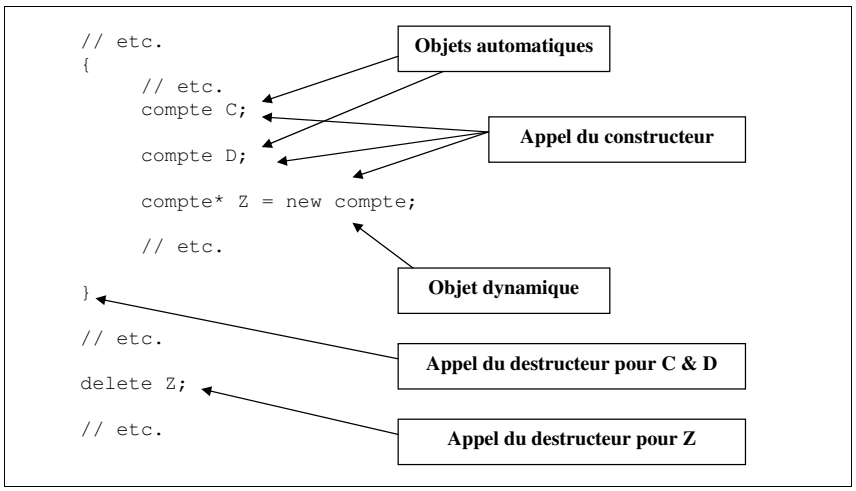
2. Durée de vie d'un objet

- Un objet a une vie

- 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.



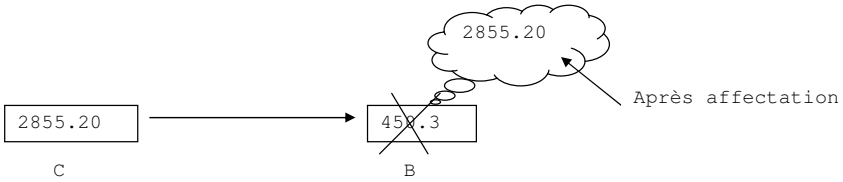
4. Affectation

- Par défaut, affectation membre par membre.

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

compte C(2855.20);
compte B(450.3);

B = C; // affectation.
```



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);

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

- Ou bien

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

- Attention : il faudra différencier entre recopie et affectation.

```
compte B; // création de l'objet.
B = C;    // pas de recopie mais uniquement l'affectation
          // car l'objet est 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 {
    // on suppose que nous avons un ptr sur un seul élément.
    int *ptr;
    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() {
    // appel du constructeur
    // déclaration permise car par défaut: d=1000.
    test 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.
    test y = x;
    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 recopie 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 il a été 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;
    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;

    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`. Ce dernier doit appeler le constructeur de `compte`.
- En premier, ce sont les membres « données » de `compte` qui sont initialisés puis ce sera le tour des autres membres "données" de `client`.

7. Classe canonique

On appelle une classe canonique toute classe qui contient au moins les éléments suivants :

- Au moins un constructeur régulier
- Un constructeur de recopie
- Un destructeur
- Un opérateur d'affectation

```
class bidon {
   // membres privés
   public:
   bidon(…); // un constructeur régulier
   bidon(const bidon&); //un constructeur de recopie
   ~bidon(); // un destructeur
   bidon &operator=(const bidon&); // opérateur d'affectation
};
```

Il est conseillé de concevoir des classes canoniques afin de prévoir les cas où la classe dispose de pointeurs sur des parties dynamiques. Nous avons déjà étudié ces cas là, dans la première partie de ce chapitre. Dans ce qui suit, nous allons examiner ces mêmes cas pour l'opérateur d'affectation et l'intérêt d'en définir un dans une classe.

8. Opérateur d'affectation =

8.1. Généralités

Un constructeur de recopie est appelé dans 3 situations suivantes :

- Lors de la création en même temps que l'initialisation de l'objet créé;
- Passage de paramètres;
- Retour d'un objet comme résultat de fonction.

Chaque classe possède un opérateur d'affectation par défaut. L'affectation est superficielle comme la copie et consiste en une affectation (de surface) membre à membre. Ceci pose les mêmes problèmes que ceux posés par le constructeur de recopie par défaut (voir les précédents paragraphes).

Un opérateur d'affectation sert donc à l'affectation dans une expression et, retourne une référence. La signature de cet opérateur est comme suit :

```
x& operator=(x)
```

Ceci est équivalent d'écrire :

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

Le résultat de l'affectation est dans l'objet appelant.

L'opérateur d'affectation doit être une fonction membre.

8.2. Exemple d'une classe tableau

```
#include <iostream>

using namespace std;

class tableau {
   int n; // la taille du tableau tab
   double *tab; // pointeur vers un tableau de double
   void alloc_test(){ // test d'allocation mémoire
    if (tab==NULL) {
       cerr << "allocation de la mémoire a échoué !\n";
       exit(1);
    }
   }
   // test la taille du tableau
   void index_test(int taille_entree){
    if (taille_entree<1) {
       cerr << "taille du tableau à allouer est inférieure\
             ou égale à zéro! \n";
       exit(1);
    }
   }
```



```
   // recopie élément par élément
   void recopie_elt(const tableau& T) {
    for (int i=0;i<n;i++) tab[i] = T.tab[i];
   }
public:
   // constructeur
   tableau(int x) {
    index_test(x);
    tab = new double[n=x]; // allocation
    alloc_test();
   }
   // constructeur de recopie
   tableau(const tableau& T) {
    tab = new double[n=T.n];
    alloc_test();
    recopie_elt(T);
   }
   // opérateur d'affectation
   tableau& operator=(const tableau& T) {
    // on teste si on n'affecte pas l'objet à lui-même.
    if (this != &T) {
       delete [] tab; // on détruit les éléments déjà alloués.
       // on alloue de la mémoire à nouveau.
       tab = new double[n=T.n];
       alloc_test(); // test d'allocation.
       recopie_elt(T); // recopie des éléments de T.tab à tab.
    }
    return (*this); // on retourne l'objet appelant.
   }
```

```
   // initialise tous les éléments de tab à nbre.
   void init_tab (double nbre){
    for (int i=0;i<n;i++) tab[i] = nbre;
   }
   // destructeur
   ~tableau(){
    delete [] tab; // libération.
   }

   // affichage vers la sortie de tab.
   void affiche () {
    for (int i=0;i<n;i++)
       cout << "tab[" << i << "] " << tab[i] << endl;
   }
};
```



```
int main() {

   tableau a(3); // on crée un tableau de 3 éléments.
   a.init_tab(10.5); // on initialise tous ses éléments à 10.5

   cout << "on affiche a ...\n";
   a.affiche(); // on affiche le contenu de a.

   // on crée un second tableau b de taille 2 (éléments)
   tableau b(2);

   // affectation de a vers b. Suite à l'utilisation de
   // operator= la taille de b devient égale à 3.
   b = a;

   cout << "le tour de b=a ...\n";
   b.affiche(); // on affiche b.

   return 0;
}
```

```
En sortie:

on affiche a ...
tab[0] 10.5
tab[1] 10.5
tab[2] 10.5
le tour de b=a ...
tab[0] 10.5
tab[1] 10.5
tab[2] 10.5
```