Cours3 - C++

Classes et objets en C++ : partie 2

J-F. Kamp

Janvier 2025

Copy-constructeur et affectation entre objets

Copy-constructeur

Un objet peut être passé en argument à une méthode :

- par pointeur (idem Java) :

```
void maMethode ( CCercle* c1 ) { ... }
```

Ici `c1` contient l'adresse de l'objet passé en paramètre. Par l'intermédiaire de `c1`, on agit sur le MEME emplacement mémoire que l'objet du programme appelant.

- par référence (presque identique à pointeur) :

```
void maMethode ( CCercle& c1 ) { ... }
```

`c1` est le CONTENU de l'emplacement mémoire. Cet emplacement mémoire est le MEME que celui du programme appelant.

Copy-constructeur

Un objet peut être passé en argument à une méthode :

- par valeur (impossible en Java) :

```
void maMethode ( CCercle c1 ) { ... }
```

Ici c1 est une variable locale de type CCercle. Au moment de l'appel de la méthode, le cercle c1 est construit à l'identique du cercle passé en paramètre, MAIS il s'agit de deux objets DIFFERENTS en mémoire.

=> Un constructeur particulier doit permettre de construire un NOUVEL objet CCercle à partir d'un autre objet CCercle. Ce constructeur est appelé COPY-CONSTRUCTEUR (constructeur de recopie).

Copy-constructeur

Revenons à la classe CListe qui possède un attribut de type pointeur sur une zone dynamique.

```
// Déclaration à écrire dans le fichier CListe.h

class CListe {

    // Attributs d'instance
    private :
        int m_nbElem;
        // int* => allocation DYNAMIQUE avec NEW
        int* m_pTabNbres;

    // Constructeur
    public :
        CListe ( int nbElem );
    // Autre méthode publique
    public :
        int getMax ( );
    // DESTRUCTEUR
        ~CListe ();
};
```

Copy-constructeur

- Supposons un passage par valeur d'un objet de type CListe

```
void maMethode ( CListe unObj ) {
        ...
}
```

- Si aucun copy-constructeur n'est défini, C++ en prévoit un par défaut. Ce constructeur effectue une copie des attributs de premier niveau :

```
this–>m_nbElem = toCopy.m_nbElem
this–>m_pTabNbres = toCopy.m_pTabNbres
```

- Cette copie de premier niveau pose problème dès que la classe possède un attribut de type pointeur car elle NE DUPLIQUE PAS les emplacements mémoires dynamiques (recopie des adresses !!)

Copy-constructeur

- Dans le fichier “CListe.h”, on rajoute donc un copy-constructeur qui définit complètement l’opération de recopie (en particulier il duplique la zone mémoire dynamique)

  CListe (const CListe& listToCopy);

- Ce constructeur :

  - crée un NOUVEL objet CListe et initialise ses attributs par recopie du contenu de l’objet CListe passé par référence (&)

  - cet objet passé en paramètre ne sera pas modifié par la méthode (const)

Copy-constructeur

Dans le fichier “CListe.cpp”, on précise le code du copy-constructeur

```
// NE PAS OUBLIER d’inclure le fichier de déclaration
#include "CListe.h"
...
// Code du copy-constructeur
CListe :: CListe ( const CListe& listToCopy ) {

    // !! this est de type pointeur sur le nouvel objet
    // qui vient d’être construit
    this –> m_nbElem = listToCopy.m_nbElem;
    this –> m_pTabNbres = new int[this –> m_nbElem];

    for ( int i=0; i < this –> m_nbElem; i++ ) {

        m_pTabNbres[i] = listToCopy.m_pTabNbres[i];
    }
}
```

Affectation d'objets

- En Java, le symbole d'affectation (=) entre 2 objets de même type signifie DANS TOUS LES CAS : recopie des adresses

- En C++, le symbole d'affectation entre 2 objets de même type signifie "recopie des adresses" SI ET SEULEMENT SI les 2 variables sont de type POINTEUR sur des objets

```
CCercle* c1 = new CCercle ( 12, -25, 10 );
CCercle* c2 = new CCercle ( 23, 89, 14 );

// Destruction obligatoire de l'espace
// mémoire pointé par c2 AVANT affectation
delete c2;
// Recopie d'adresses
c2 = c1;
// c1 ET c2 pointent sur le MEME objet
```

Affectation d'objets

- En C++, le symbole d'affectation (=) entre 2 objets a une TOUTE AUTRE signification si les variables NE SONT PAS de type pointeur

```
CCercle c1 ( 12, -25, 10 );
CCercle c2 ( 23, 89, 14 );

// Que se passe-t-il si affectation ?
c2 = c1;
```

PAR DEFAUT, le compilateur effectue une SIMPLE recopie des contenus des attributs de c1 dans ceux de c2 (recopie de premier niveau), ce qui est équivalent à :

```
c2.m_x = c1.m_x;
c2.m_y = c1.m_y;
c2.m_r = c1.m_r;
```

Affectation d'objets

Problème : la recopie premier niveau (celle par défaut) NE DUPLIQUE PAS les emplacements mémoires dynamiques (idem copy-constructeur)

```
// Déclaration à écrire dans le fichier CListe.h

class CListe {

    // Attributs d'instance
    private :
        int m_nbElem;
        // int* => allocation DYNAMIQUE avec NEW
        int* m_pTabNbres;

    // Constructeur
    public :
        CListe ( int nbElem );
    // Autre méthode publique
    public :
        int getMax ( );
    // DESTRUCTEUR
        ~CListe ();
};
```

Affectation d'objets

```
CListe list1 (50);
CListe list2 (60);

list2 = list1; // Que se passe-t-il ?

// Ceci par défaut :
list2.m_nbElem = list1.m_nbElem;
list2.m_pTabNbres = list1.m_pTabNbres;
```

Problèmes :

1. m_pTabNbres étant de type pointeur, list1.m_pTabNbres et list2.m_pTabNbres pointent sur le MEME emplacement mémoire

2. conséquence (souvent oubliée) : à la disparition de list1 et list2, le destructeur de chacun des objets est appelé => le même emplacement mémoire est détruit 2 fois !!

Affectation d'objets

- Solution : il faut absolument (sauf cas exceptionnels) effectuer une RECOPIE des emplacements mémoires dynamiques pour avoir une réelle duplication lors de l'affectation

- IL FAUT donc (re)spécifier l'opération "=" pour la classe CListe : cette opération de définition de l'opérateur "=" pour la classe CListe s'appelle SURCHARGE (surdéfinition) d'opérateurs

Affectation d'objets

- Dans le fichier "CListe.h", on rajoute la méthode suivante qui précise en quoi consiste l'opération "=" pour cette classe

```
void operator= ( const CListe& listToCopy );
```

- Cette méthode :
  - précise qu'elle surcharge l'opérateur **=**
  - elle prend en paramètre l'objet à copier listToCopy passé par référence (&)
  - cet objet passé en paramètre ne sera pas modifié par la méthode (const)
  - elle ne renvoie rien

Affectation d'objets

Que faut-il écrire pour le code de la méthode ?

```
void operator= ( const CListe& listToCopy );
```

```
CListe list1 (50);
CListe list2 (60);
```

Dans l'affectation : `list2 = list1;`
Qui doit être modifié ? `list2`
Qui reste inchangé ? `list1`

DES LORS :

- `void operator= ( ... )`
  est méthode d'instance de `list2`
- `list1` est passé en paramètre à cette méthode
- `list2 = list1` est équivalent à

```
list2.operator= ( list1 );
```

Affectation d'objets

```
#include "CListe.h"
...
// La méthode de définition de l'opération « = »
// Si opération « list2 = list1 », « list2 » correspond à
// « this » tandis que « list1 » est passé par référence
// dans listToCopy
void CListe :: operator= ( const CListe& listToCopy ) {

    // !! this == &list2
    // !! listToCopy : c'est l 'objet « list1 » lui-même

    if ( this != &listToCopy ) {
        if ( m_pTabNbres != NULL ) delete[] m_pTabNbres;
        this –> m_nbElem = listToCopy.m_nbElem;
        this –> m_pTabNbres = new int[this –> m_nbElem];

        for ( int i=0; i < this –> m_nbElem; i++ ) {
           m_pTabNbres[i] = listToCopy.m_pTabNbres[i];
        }
    }
}
```

Ca ressemble FORTEMENT au code du copy-constructeur.

Affectation d'objets

Si l'on veut pouvoir effectuer la double affectation suivante :

```
CListe list1 (50);
CListe list2 (60);
CListe list3 (70);

list1 = list2 = list3;
```

Ce qui est équivalent à :

```
list1.operator= ( list2. operator= ( list3 ) );
```

Il faut que la méthode retourne `CListe&`

```
CListe& operator= ( const CListe& listToCopy );
```

Et la dernière instruction de la méthode est :

```
return *this;
```

Conclusion

Dorénavant, dès qu'une classe CT contient au moins un attribut de type pointeur, il faudra OBLIGATOIREMENT définir dans cette classe (REGLE des TROIS en C++) :

- un destructeur :

  ~CT ();

- un constructeur de recopie :

  CT (const CT& unObj);

- une surcharge de l'opérateur d'affectation :

  CT& operator = (const CT& unObj);

- règle de bonne pratique :

  Toujours allouer les pointeurs ou les mettre à NULL dès la construction