
TD 4

Opérations de copie

4.1 Objectif

Cette série d'exercices vise à montrer l'importance qui doit être accordée à la définition correcte des opérations de copie en C++ : constructeur de copie et opérateur de copie à l'affectation.

4.2 Synthèse automatique des opérations de copie par C++

On considère la classe A dont la définition complète est la suivante (fichier Copy/A.h) :

```
class A {
private:
    string _s;
    int _i;
public:
    A(const string& s = "", int i = 0)
    {
        _s = s;
        _i = i;
    }
};
```

Est-ce que les instructions suivantes (fichier Copy/main_A.cpp) compilent correctement ?

Si oui, pourquoi ? Si non, pourquoi ?

```
A a1;
A a2("hello");
A a3 = a2;
a1 = a2;
string s = "bonjour";
a2 = s ;
a2 = "bonjour";
```

On modifie légèrement la définition des membres de données de la classe A :

```
class A {
private:
    const string _s;
    int _i;
    // ...
};
```

et on demande de répondre aux mêmes questions.

4.3 Importance des opérations de copie

Dans le répertoire List, on trouvera dans les fichiers List.h et List.cpp la définition complète d'une liste simplement chaînée d'entiers. Cette liste possède les trois opérations de base

`append()` (ajouter un entier à la fin de la liste), `prepend()` (ajouter un entier au début) et `get_first()` (retirer le premier élément de la liste et retourner sa valeur). Le contenu de la liste est également imprimable grâce au maintenant habituel `operator<<`. Enfin, le fichier `main_List.cpp` fournit un programme de test de la liste.

4.3.1 Expérimentation avec les destructeurs et les copies

1. Allez sous le répertoire `List` recopié chez vous lors du premier TD.
2. Compilez le programme de test (il suffit de faire `make`) et exécutez le (l'exécutable se nomme `tst_List.exe`). Tout doit bien se passer.
3. La classe `List` fournie n'a pas de destructeur. Modifiez la pour lui en donner un, qui détruit toutes les cellules de la liste. Compilez et exécutez le programme de test. Tout devrait continuer à bien se passer.
4. Instrumentez votre destructeur pour déterminer combien de fois il est invoqué par le programme de test. Justifiez le résultat.
5. Éditez le fichier `main_List.cpp` et décommentez les 4 lignes précédées par
// First, uncomment the following 4 lines
Compilez et exécutez à nouveau. Il se peut que le programme crashe. Même si ce n'est pas le cas, le résultat est-il alors correct ?
6. Commentez à nouveau les 4 lignes précédentes (pour éviter de crasher de la même manière) et décommentez la ligne précédée par
// Second, uncomment the next 3 lines
Compilez et exécutez à nouveau. Il se peut que le programme crashe une seconde fois. Même si ce n'est pas le cas, le résultat est-il alors correct ? Au passage, combien de fois est appelé le destructeur dans ce cas ?
7. Analysez les causes des deux crashes précédents.
8. Modifier en conséquence la définition de la classe `List` pour que le programme de test ne crashe plus (une fois les deux zones décommentées).

Note Dans tout cet exercice, vous ne devez pas modifier le fichier `main_List.cpp` autrement que pour commenter et décommenter les zones indiquées.

4.3.2 Pour ne pas perdre la main...

En vous inspirant de ce que nous avons fait lors des premiers TDs avec les classes `Stack` et `Fifo`, transformez la classe `List` pour qu'elle devienne générique (*template*) : au lieu d'une liste d'entiers, on définit donc une liste de « n'importe quoi », pourvu qu'il soit copiable.

Attention Dans cette partie de l'exercice, il y a quelques subtilités techniques. Tout d'abord, rappelez-vous que, dans le cas des classes templates, la différence entre `.h` et `.cpp` devient illusoire : soit on met tout dans le `.h`, soit on inclut le `.cpp` à la fin du `.h`, et on modifie la `Makefile` afin de ne pas compiler séparément le `.cpp` (voyez comment cela est fait dans `Stack`). Ensuite, il se peut que vous ayez besoin d'aide pour traiter correctement la fonction amie `operator<<`.