Travaux Dirigés Programmation C++ : Feuille 5

Informatique 2ème année.

—*Julien Allali* - allali@enseirb.fr —

Ce dernier TD est une introduction à la programmation générique.

1 Transformer une classe en classe générique

Vous trouverez dans l'archive suivante : smart.tar une implémentation des `SmartPointer` pour la gestion d'un entier.

Un `SmartPointer` est une classe dont les instances se comportent comme des pointeurs. Dans cet objectif, la classe redéfinit par exemple les opérateurs `*` et `->` permettant d'accéder à la valeur pointée (en fait, contenue dans l'instance). En plus de gérer une zone mémoire, les `SmartPointer` conservent un compteur associé à cette zone et libère celle-ci si plus personne ne l'utilise.

Dans la classe vous trouverez deux attributs : `data` qui pointe vers la donnée prise en charge par le `SmartPointer` et `counter` qui compte le nombre de `SmartPointer` pointant vers cette même donnée.

La classe qui vous est fournie correspond à des `SmartPointer` pour des entiers.

▸ **Exercice 1.** `SmartPointer` *générique :*

Le premier exercice consiste à faire en sorte que les `SmartPointer` *puissent être utilisés pour tout type de donnée (et pas uniquement des* `int`*). Nous allons procéder étape par étape, veiller à bien suivre les indications et tout se passera bien :).*

1. *Mettre le corps des méthodes dans le fichier* `.hpp` *: Pour chaque méthode de la classe* `SmartPointer`*, placer le code de la méthode dans le fichier* `.hpp`*. Une fois ceci réalisé, vérifier que tout se passe bien lorsque l'on compile le fichier* `SmartExample.cpp` *(*`g++ Wall SmartExample.cpp`*-).*
2. *Introduire un type variable pour la classe* `SmartPointer` *: faire la modification suivante au début de la déclaration de la classe*

   ```
   template<class DataType>
   class SmartPointer{
   ```

   Une fois cette modification réalisée, votre classe dépend maintenant d'un type variable nommé `DataType` *dans la classe. Pour utiliser cette classe paramétrée, vous devez indiquer le type qui doit être utilisé pour* `DataType`*.*

   ```
   SmartPointer<int> p(new int(10));
   ```

   modifier le fichier `SmartExample` *pour prendre en compte cette modification et tester le programme.*
3. *Utilisons le type générique dans la classe : l'objectif maintenant est d'utiliser le type* `DataType` *dans la classe. Modifier l'ensemble de la classe en remplaçant tous les* `int` *relatifs à l'attribut* `data` *par* `DataType`*. Une fois ceci réalisé, vérifier que votre programme d'exemple compile toujours.*
4. *Et avec d'autres types ? Vérifier que votre classe* `SmartPointer` *est bien générique en l'utilisant avec d'autres types (primitifs ou utilisateurs). En particulier, on testera le fonctionnement avec la classe* `Chaine` *fournie dans le TD précédent. Après avoir ajouté l'affichage d'un message dans le destructeur de la classe* `Chaine`*, vérifier que celui-ci est bien appelé lors de la libération du dernier pointeur utilisant l'instance.*

   ```
   int main(){
     SmartPointer<Chaine> p=new Chaine("toto");
     SmartPointer<Chaine> q=p;

     printf("La chaine est: %s, de taille %d",(const char *)(*p),p->taille());
   }
   ```
5. *Et un problème pour finir : Tester (avec valgrind/gdb par exemple) la validité du code suivant :*

   ```
   int main()
   {
     SmartPointer<int> buggy(new int[10]);
   }
   ```

   Vous expliquerez le comportement observé sans chercher à le corriger (ce sera fait plus loin).

▸**Exercice 2.** *Utilisation des* `SmartPointer<char>` *dans* `Chaine` *:*
Dans la classe `Chaine` *utiliser les* `SmartPointer` *pour l'attribut* `_donnees`*. Est-il raisonnable que les instances de la classes* `Chaine` *partagent la chaîne de caractères interne ?*

Effectuer l'ensemble des modifications nécessaire. En se basant sur la dernière observation de l'exercice précédent, modifier le mode de libération de l'attribut `_data`*.*

Observer attentivement le constructeur par recopie, le destructeur et l'opérateur d'affectation. Supprimer le code superflu.

▸**Exercice 3.** *Le problème de la libération des ressources :*
Comme nous l'avons observé dans le premier exercice, il y a un problème pour la libération des ressources. En effet le `SmartPointer` *a la charge de libérer des ressources qu'il n'a pas lui même allouées. Donner un diagramme de classes permettant de résoudre ce problème en utilisant un lien* a un*.*

Nous allons mettre en place une solution reposant sur les templates. Pour cela nous allons ajouter un nouveau paramètre générique à la classe `SmartPointer`*. Nous avons alors deux solutions :*

— *Tout d'abord, il est possible de paramétrer une classe à l'aide de constantes pour les types primitifs :*

```
template<int n>
struct EntierConstant{ static int value() const {return n;} };

EntierConstant<5>.value();
```

Ceci est possible avec tous les types primitifs, entre autres les pointeurs de fonctions.
Dans la déclaration de la classe `SmartPointer`*, ajouter un paramètre générique* `FreeFunction` *dont le type est un pointeur de fonction qui retourne* `void` *et qui prend un pointeur de type* `T` *en argument.*
Dans le fichier `SmartExample`*, ajouter une fonction prenant un pointeur d'entiers en argument et libérant celui-ci à l'aide de l'opérateur* `delete`*. Ajouter une autre fonction effectuant la libération à l'aide de* `delete[]`*. Reprendre l'exemple du premier exercice avec le* `buggy` *et faire en sorte que tout marche bien.*

— *L'autre solution consiste à ajouter comme deuxième paramètre template une classe. Lors de la libération il suffit alors d'instancier un objet de cette classe et, par exemple, d'utiliser l'opérateur de fonction sur cet objet :*

```
template<class DataType,class FreeFonctor>
class SmartPointer{ ...
  void releasePointer(){
    ...
    FreeFonctor f;
    f(data);
    ...
  } ...
};
```

Nous garderons la solution précédente dans notre code.

▸**Exercice 4.** *Libérateur générique et valeur par défaut :*
La solution mise en œuvre précédemment implique d'écrire une fonction de libération pour chacun des types utilisés avec les `SmartPointer`*. Or, dans la plupart des cas, la libération se fait avec* `delete` *ou encore* `delete[]`*. Plutôt que d'écrire de nombreuses fois la même fonction avec juste le type du paramètre qui change, nous pouvons écrire une fonction template pour chaque mode de libération.*

Dans le fichier `SmartPointer.hpp`*, ajouter dans l'espace de nom* `enseirb` *une fonction générique* `freeWithDelete` *qui procède à la libération d'un pointeur passé en argument à l'aide de l'operateur* `delete`*. On écrit ensuite la fonction* `freeWithDeleteArray` *qui fait de même mais pour les allocations de tableau.*

Modifier le fichier `SmartExample` *pour utiliser ces nouvelles fonctions.*

Dans le cas général, nous souhaitons que ce soit l'opérateur `delete` *qui soit utilisé pour libérer la ressource. Pour que dans ce cas, l'utilisateur n'ait pas à spécifier la fonction de libération, modifier la déclaration de la classe* `SmartPointer` *pour que la fonction* `freeWithDelete` *soit utilisée par défaut pour le paramètre* `FreeFunction`*.*

Pour finir, modifier la classe `Chaine` *pour prendre en compte ces modifications.*

▸**Exercice 5.** *Rope :*
Les `Rope` *sont des chaînes de caractères améliorées optimisant le partage mémoire et la copie paresseuse.*

Dans le cas de chaînes constantes, le problème est plus simple car on peut partager les ressources sans avoir à se poser de question sur l'intégrité de celles-ci.

Modifier la classe `Chaine` *en ajoutant une méthode* `sub(int b,int e)` *qui renvoie une nouvelle* `Chaine` *représentant la sous chaine contenue entre les positions* `b` *et* `e`*. Faire en sorte que la chaîne interne (mémoire) soit partagée entre les deux chaînes. Pour cela, ajouter des attributs dans la classe chaine indiquant où sont le début et la fin de la chaine par rapport à l'attribut* `_donnees`*.*

Dans le même esprit, ajouter l'opérateur `+` *effectuant la concaténation de deux chaînes. Toujours pour optimiser le partage de la mémoire, modifier une fois de plus le codage interne de la classe Chaine afin qu'une*

chaine soit maintenant une liste chainée de blocs. Un bloc correspond à un `SmartPointer`*, une position de début et une taille. Pour l'implémentation de cette liste chaînée, vous utiliserez le type* `slist` *de la STL (vous trouverez de la documentation ainsi que des exemples d'utilisation sur la page* `http://www.sgi.com/tech/stl/`*).*