Demi-Projets de C++

Julien Allali : allali@univ-mlv.fr

1 Déroulement.

Ce projet se compose de trois sujets au choix, ceux-ci sont présentés dans les sections suivantes. L'ensemble des étudiants de la promotion doivent se répartir sur ces trois sujets, c'est pourquoi vous devez vous inscrire sur une feuille de choix au niveau des panneaux d'affichage à partir du 10 Février 13h.

Une fois votre sujet choisi, vous disposez au minimun de 4 semaines pour le réaliser **seul(e)**. Puis, lors d'un TP d'évaluation vous aurez une heure pour ajouter une fonctionnalité à votre projet.

IMPORTANT: Dans la suite, vous devez **impérativement** respecter les noms des classes, méthodes, espaces de noms, ... **avec la case**, de même pour les noms des fichiers (évidement, le non respect de cette régle entraîne un zero).

Enfin, sur le site http://monge.univ-mlv.fr/~allali/ dans la section Enseignement, une page concernant ces projets a été mis en place. Vous trouverez entre autre sur cette page, des testes qui doivent obligatoirement avoir été validé pour pouvoir se présenter au TP noté.

2 SmartPointer

2.1 Introduction et définition

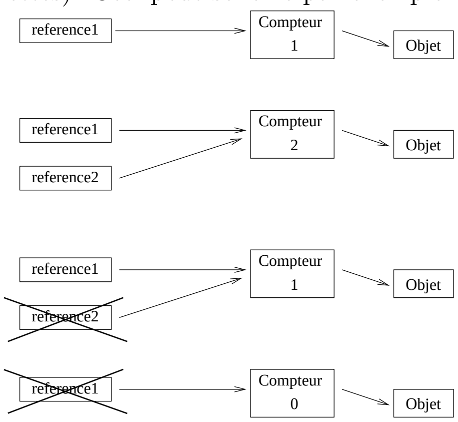
Un *Smart Pointer* est un objet fournissant les mêmes fonctionnalités qu'un pointeur plus certaines autres. De ce fait, un *Smart Pointer* a pour principal objectif de permettre d'accéder à un objet (via l'opérateur de déréférencement entre autre).

Parmi les fonctionnalités pouvant être proposées par des *Smart Pointer* on peut avoir: la gestion de ressource mémoire (désallocation automatique), la copie paresseuse, une meilleure gestion du pointeur *NULL* (exceptions) etc...

Un exemple simple de *Smart Pointer* est la classe *auto_ptr* de la librarie standard C++. Un objet *auto_ptr* pointe sur un objet alloué dynamiquement. Lorsqu'il est détruit, il désalloue l'objet sur lequel il pointe. Lorsqu'on copie un *auto_ptr* dans un autre, l'objet interne est transferé, ainsi il n'y a qu'un seul *auto_ptr* par objet.

Mais si les *Smart Pointer* sont connus, c'est surtout pour permettre la mise en place de systèmes de désallocation automatique de la mémoire (ramasse miettes). Ceci peut se faire par exemple de la manière suivante:

```
void f(SmartPointer ref2)
{
...
}
...
int main(void)
{
SmartPointer ref1(new Objet());
.....
f(ref1);
.....
return 0;
}
```



1 A chaque objet dynamique, on attache un entier indiquant le nombre de références sur l'objet.

2 Si le nombre de références sur un objet tombe à 0, alors l'objet est libéré.

2.2 Ce qu'il faut faire:

Vous devez implémenter dans un fichier **SmartPointer.hpp** une classe **SmartPointer** faisant parti de l'espace de nom **utils**. Cette classe générique dépendra d'un type T correspondant au type du pointeur qui sera contenu dans la classe. Cette classe doit implémenter le concept de compteur de référence (décrit plus haut). Les opérateurs `*`, `->`, `=` ainsi que ceux de conversion devront être écrits.

3 Matrice Générique

Dans un fichier **Matrix.hpp**, vous devez écrire une classe générique **Matrix** faisant partie d'un espace de nom **umlv**. Les opérateurs `+`,`-`,`+=`,`-=`, `*`, `*=`,`=` devront être écrits. Enfin, vous devez gérer vous même la mémoire dans la classe (interdiction d'utiliser les conteneurs de la STL).

4 Sac

Un sac est un conteneur pouvant contenir zéro ou plusieurs objets, chaque objet n'étant pas forcément unique. Un objet peut donc être placé plusieurs fois dans le sac. Un sac dispose de deux fonctions permettant d'y mettre et d'en retirer un objet. Il a aussi une fonction permettant de dire si un objet se trouve dans le sac. Ainsi, un sac contiendra des couples (o,n) (vous pouvez utiliser le type std::pair) indiquant que n objets o sont présents dans le sac.

Vous devez implémenter dans un fichier **Bag.hpp**, la classe **Bag** faisant partie de l'espace de nom **container**. Hormis la classe std::pair, vous ne devez pas avoir recours aux classes de la STL. les fonctions suivantes devront être implémentées:

- operator +: concaténation de deux sacs.
- operator -: S1 - S2: retire de S1 les objets de S2.
- add(o,n=1): ajoute n fois l'objet o dans le sac.
- remove(o,n=1): retire n fois l'objet o du sac.
- find(o): nombre d'occurence de o dans le sac.

Voici un exemple d'utilisation de la classe Bag:

```
Bag<char> b;   // b est vide
b.add('c',1);  // b = { ('c',1) }
b.add('c',2);  // b = { ('c',3) }
b.add('a',2);  // b = { ('c',3) ; ('a',2) }
cout<<b.find('a')<<endl // affiche 2
cout<<b.find('b')<<endl // affiche 0
b.remove('a',1);  // b = { ('c',3) ; ('a',1) }
cout<<b.find('a')<<endl // affiche 1
```