Travail Pratique 4
Date de remise : 4 mai 2018 23h55
Ce travail peut être fait individuellement ou en équipe de 2

Conception et programmation orientée objet II
420-401-AL

1 Objectifs

— Gestion de la mémoire
— Création d'objets
— Surcharge des opérateurs
— Application du concept de programmation générique

2 Présentation du problème

La gestion de la mémoire est un problème auquel nous devons porter une attention très particulière quand on programme dans un langage comme C++. Une erreur d'inattention peut faire en sorte que l'on perd des heures à chercher la source d'un bug. Dans ce travail, vous devrez créer une implantation simplifiée d'une classe `SmartPointer` qui vous permettra d'éviter que vous effaciez deux fois le même pointeur.

3 Description du fonctionnement de la classe

Le principe d'un `SmartPointer` est simple. C'est une classe qui a comme variable membre un pointeur, pointant vers un objet, tableau ou variable de type primitive. La classe contient aussi un compteur permettant de conserver le nombre de référence à l'emplacement mémoire. Ainsi, on n'appliquera pas de destruction de l'emplacement mémoire tant et aussi longtemps que personne ne l'utilise (plus). Voici un exemple de situation problématique que cette stratégie permettra d'éviter :

```
int *tab1 = new int[10];
int *tab2 = tab1;

delete[] tab2;
delete[] tab1;
```

Dans cet exemple, on efface deux fois le même emplacement mémoire, ce qui conduira à une erreur de segmentation. Si on avait compté le nombre de références à l'emplacement mémoire de tab1, on aurait vu que deux références sont faites à cet emplacement, et on n'aurait pas libérer la mémoire tant et aussi longtemps que quelque chose pointe encore à cet endroit.

Ce type de stratégie permettra ainsi d'éviter que des erreurs surviennent lors de la copie d'objets contenant des pointeurs comme variable membre. La destruction d'une copie n'entraînera pas la destruction de ses pointeurs, dans le cas où un autre objet contient aussi les mêmes pointeurs.

Ainsi, à aucun moment dans votre code, on ne devrait manipuler directement un pointeur, mais toujours des pointeurs intelligents déclarés sur la pile contenant des pointeurs vers des endroits spécifiques du tas.

3.1 Détails d'implantation

Notre but est d'implanter une classe qui permettra la gestion de la mémoire pour des pointeurs de n'importe quel type. Ainsi, nous utiliserons les principes de la programmation générique. De plus, notre classe devra simuler un pointeur pour l'utilisateur, donc nous devrons redéfinir les opérateurs relatifs aux pointeurs en C++. Vous trouverez dans ce qui suit une description sommaire de la classe **SmartPointer** que vous devez implanter.

3.1.1 Variables membres

Les types des variables ne sont pas spécifiés, à vous de les définir pour votre implantation.

- `mPtr` : un pointeur vers l'emplacement mémoire qui est "gardé" par l'instance `SmartPointer`
- `mNbRef` : un pointeur sur un entier compteur du nombre de références à `mPtr`
- `mDeleteFonction` : un pointeur vers une fonction statique de destruction du pointeur `mPtr`. En effet, si `mPtr` est un tableau, il faut le détruire comme suit :

  ```
  delete[] mPtr;
  ```

  Sinon, on le détruit comme suit :

  ```
  delete mPtr;
  ```

  Voici les deux fonctions **statiques** à inclure :
  - `void deletePtr(T* ptr)`
  - `void deleteTab(T* ptr)`

  Ainsi, lors de la construction d'un SmartPointer, un usager peut spécifier que son pointeur devra être détruit avec une ou l'autre des deux fonctions. Il spécifiera son choix en passant en argument un pointeur de fonction vers l'une ou l'autre des deux fonctions de destructions statiques de la classe. Le pointeur de fonction `mDeleteFonction` pointera alors vers l'une ou l'autre des ces deux fonctions statiques, permettant ainsi

à l'usager d'avoir un pointeur intelligent englobant soit un tableau, ou un pointeur individuel.

3.1.2 Les fonctions à définir

Encore une fois, le type de retour et de paramètre n'est pas donné, et vous pouvez ajouter des fonctions, comme par exemple :

- Un constructeur vide
- Un constructeur prenant les arguments suivants :
  1. un pointeur vers la donnée à englober par votre SmartPointer
  2. un pointeur vers une fonction de destruction
- Un constructeur par copie : je vous rappelle qu'un constructeur par copie est invoqué lors de déclaration du type C c1(c2) et C c1=c2, lors du passage d'arguments par valeurs et du retour de fonction par valeur.
- Un destructeur : je vous rappelle qu'un destructeur est appelé lors du retour de fonction par valeur, quand on utilise **delete** sur un objet ou quand une variable n'est plus dans la portée courante.
- Des opérateurs `=`, `->`, `*` et `[]`pour votre `SmartPointer`
- La fonction `getCount()`, qui vous donnera accès au nombre de références à `mPtr`. Cette fonction pourra être utilisée durant le débogage avec "assert".

Voici une définition qui pourra être utile pour la déclaration de vos pointeurs sur les fonctions statiques :

```
typedef void(*DeleteFunctionType)(T *p);
```

Cela définit le type **DeleteFunctionType** comme un pointeur sur une fonction **void** prenant en argument un pointeur de T, exactement le type des fonctions **deletePtr** et **deleteTab**.

4 Exemple d'utilisation de la classe

```
int main(int argc, char *argv[])
{
        //TEST LE COMPTE DE REFERENCES
        SmartPointer<int> sm(new int(4));
        SmartPointer<int> sm2 = sm;
        assert(sm2.getCount() == 2 && sm.getCount() == 2);
        SmartPointer<int> sm3;
        SmartPointer<int> sm4;
        // TEST surcharge de =
        sm3 = sm2;
        assert(sm3.getCount()==3 && sm2.getCount()==3);
        assert(sm.getCount()==3);
        sm4 = sm2;
        assert(sm4.getCount()==4 && sm3.getCount()==4);
        assert(sm2.getCount()==4 && sm.getCount()==4);
        //TEST LE DESTRUCTEUR
        SmartPointer<std::string> smartPS3;
        {
                SmartPointer<std::string> smartPS(new string("bonjour"));
                SmartPointer<std::string> smartPS2 = smartPS;
```

```
                smartPS3 = smartPS2;
        }
        assert(smartPS3.getCount() == 1);
        //TEST surcharge de *
        assert(*smartPS3 == "bonjour");
        assert(*sm4 == 4);
        *sm4 = 3;
        assert(*sm3 == 3);
        //TEST surcharge de ->
        int length = smartPS3->size();
        assert(length == 7);
        //TEST surcharge de []
        SmartPointer<int> tab(new int[7], SmartPointer<int>::deleteTab);
        tab[0] = 0;
        cout << tab[0] << endl;
        assert(tab.getCount()==1);
}
```

5 Les pointeurs de fonctions

Les fonctions, comme les variables, ont une adresse mémoire. Ainsi, il est possible d'avoir un pointeur pointant non seulement vers une variable mais aussi une fonction. Nous pouvons ensuite appeler la fonction pointée via le pointeur de fonction. Le bout de code suivant affichera 35.

```
int max(int a, int b)
{
        if (a<b)
                return b;
        else
                return a;
}

int main()
{
        int (*ptrFoncMax) (int,int);
        ptrFoncMax = max;
        cout << ptrFoncMax(2,3);
        typedef int (*UnType) (int,int);
        UnType ptrFoncMax2;
        ptrFoncMax2 = max;
        cout << ptrFoncMax2(2,5);
}
```

6 Barême de correction

Qualité du code	**/15**
commentaires (commentaires utiles et clairs)	/5
clareté du code (nom des variables, indentation...)	/10
Fonctionnement	**/85**
Les constructeurs vide	/6
Un constructeur par copie	/15
Un operateur =	/15
Un destructeur	/15
L'opérateur `->`	/6
L'opérateur `*`	/6
L'opérateur `[]`	/6
La fonction `getCount()`	/6
Un `main` permettant de tester votre classe	/10

7 Remise

La remise sera faite dans l'environnement MOODLE, dans la section prévue à cet effet, avant le 4 mai 2018 à 23h55.