Exercice 23 - Comp'UT

Remarque : Le but de ce TD est de se familiariser avec les notions d'association, d'agrégation et de composition entre classes et leurs conséquences au niveau de l'implémentation des classes, avec notamment l'éventuel besoin de la surcharge d'un destructeur, d'un constructeur de recopie et/ou d'un opérateur d'affectation.

Remarque : Ce TD utilise le thème du projet de ce semestre afin de commencer à vous familiariser avec les différentes entités de l'application qui sera à développer. On ne perdra pas de vue que les questions développées dans ce TD ne constituent pas une architecture pour le projet. Celle-ci devra être retravaillée en tenant compte de l'ensemble des entités du sujet de projet.

On souhaite développer l'application Comp'UT, un calculateur scientifique permettant de faire des calculs, de stocker et de manipuler des variables et des programmes, et utilisant la notation RPN (Reverse Polish Notation), *c.-à-d.* la « notation polonaise inversée » dite aussi « la notation postfixe ». La notation postfixe est une méthode de notation mathématique permettant d'écrire une formule arithmétique sans utiliser de parenthèse. Pour cela, un opérateur est toujours placé après ses arguments. Ainsi :

- l'expression « `1+1` » en notation infixe peut s'écrire « `1 1 +` » en notation postfixe ;
- l'expression « `2*2+1` » en notation infixe peut s'écrire « `2 2 * 1 +` » en notation postfixe ;
- l'expression « `(2+3)*4` » en notation infixe peut s'écrire « `2 3 + 4 *` » en notation postfixe.

La notation postfixe implique l'utilisation d'une pile pour stocker les résultats intermédiaires lors de l'évaluation d'une expression. Pour intéragir avec un tel calculateur, l'utilisateur entre une expression constituée d'une suite d'opérandes. Une opérande est soit une littérale soit un opérateur. Une suite d'opérandes est évaluée de la gauche vers la droite de la façon suivante :

- si l'opérande est une littérale, elle est empilée sur la pile ;
- si l'opérande est un opérateur d'arité n, elle provoque le dépilement de la pile de n expressions, l'application de l'opérateur sur ces expressions, et l'empilement du résultat.

Dans ce TD, la calculatrice ne traitera que des *expressions* entières sur lesquelles on pourra appliquer les opérateurs binaires (*c.-à-d.* d'arité 2) suivants : l'addition, la soustraction, la multiplication et la division entière. Cette application sera ultérieurement amenée à évoluer pour traiter des expressions plus complexes comme des réels, des fractions, des complexes, des variables ou des expressions composées d'expressions plus simples ainsi que l'application d'opérateurs plus évolués et de programmes de calcul.

Dans la suite, une expression sera représentée par un objet de la classe `Expression` qui comporte un attribut `value` de type **`int`**. L'**unique constructeur** de cette classe a exactement 1 paramètre permettant d'initialiser l'attribut d'un objet. La classe possède aussi un accesseur `getValue()` qui permet de connaître la valeur de l'attribut d'un objet, ainsi qu'une méthode `toString()` renvoyant une valeur de type `string` représentant la valeur d'un objet `Expression` sous forme de chaîne de caractères.

Les objets `Expression` sont gérés et contenus par un module appelé `ExpressionManager` qui est responsable de leur création (et destruction). La classe possède un unique constructeur sans argument. Initialement, un objet `ExpressionManager` ne contient aucun objet `Expression`. La classe `ExpressionManager` possède aussi une méthode `addExpression()` qui permet de créer une nouvelle expression en transmettant une valeur de type **`int`**. Cette méthode renvoie une référence sur l'objet `Expression` qui a été créé. De plus, la classe `ExpressionManager` possède une méthode `removeExpression()` qui permet d'éliminer l'objet `Expression` dont la référence est transmise en argument.

Dans l'application, un objet de la classe `Pile` est chargé de gestion de la pile d'affichage des expressions ainsi que des messages destinés à l'utilisateur. Pour cela, un tel objet gère une pile d'objets `Item`. Un objet `Item` encapsule un attribut `exp` de type `Expression*` qui contient l'adresse d'un objet `Expression` à afficher. La classe `Item` possède un unique constructeur sans argument qui initialise l'attribut `exp` au pointeur nul. La méthode `setExpression()` permet de modifier l'attribut `exp` avec l'adresse de l'objet `Expression` qui lui est transmis en argument par référence. La méthode `getExpression()` permet d'obtenir une référence sur l'objet associé. La méthode `raz()` permet de réinitialiser l'attribut `exp` avec le pointeur nul.

Dans l'application, un objet de la classe `Controleur` est chargé de la communication avec l'utilisateur. La méthode `executer()` permet de lancer le dialogue utilisateur. La méthode `commande` qui prend un paramètre de type **`const`** `string&` permet d'interpréter une suite de caractères entrée par un utilisateur. Cette méthode assure la demande de création ou d'élimination d'expressions dans un objet `ExpressionManager` et ainsi que leur affichage dans un objet `Pile`. Les objets `ExpressionManager` et `Pile` associés à l'objet `Controleur` sont au moment de sa création.

Préparation : Créer un projet vide et ajouter trois fichiers `computer.h`, `computer.cpp` et `main.cpp`. Définir la fonction principale `main` dans le fichier `main.cpp`. Après avoir recopié le code ci-après dans le fichier `computer.h`, s'assurer que le projet compile correctement. Au fur et à mesure de l'exercice, on pourra compléter la fonction

principale en utilisant les éléments créés. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `computer.h`).

```
#ifndef _COMPUTER_H
#define _COMPUTER_H
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

class ComputerException {
 string info;
public:
 ComputerException(const string& str):info(str){}
 string getInfo() const { return info; }
};
#endif
```

computer.h

On fera attention dans chacune des questions, même si ce n'est pas précisé, de prendre en compte les éventuelles situations exceptionelles.

Question 1

Lire le sujet en entier et identifier les différentes classes de l'application. Identifier les associations qui existent entre ces classes. Quel type de lien existe t-il entre un objet `ExpressionManager` et les objets `Expression` qu'il crée et auxquels il donne accès ? Quel type de lien existe t-il entre un objet `Item` et un objet `Expression` auquel il est associé ? Quelle type d'association existe t-il entre la classe `Pile` et la classe `Item` ? Entre la classe `Controleur` et les classes `ExpressionManager` et `Pile` ?

Question 2

Définir la classe `Expression` ainsi que l'ensemble de ses méthodes. Pour définir la méthode `toString`, on pourra utiliser la classe standard `stringstream` déclarée dans le fichier d'entête `<sstream>` ou la fonction `to_string` déclarée dans le fichier d'entête `<string>`. La classe `Expression` nécessite t-elle (a priori) un destructeur, un constructeur de recopie et/ou un opérateur d'affectation ? Expliquer. Définir ces méthodes seulement si nécessaire.

Question 3

Est-il possible de définir un tableau (alloué dynamiquement ou non) d'objets `Expression` ? Expliquer. Est-il possible de créer un tableau (alloué dynamiquement ou non) de pointeurs d'objet `Expression` ? Expliquer.

Question 4

Soit la classe `ExpressionManager` dont voici une définition partielle :

```
class ExpressionManager {
 Expression** exps;
 size_t nb;
 size_t nbMax;
 void agrandissementCapacite();
public:
 ExpressionManager();
 Expression& addExpression(int v);
 void removeExpression(Expression& e);
};
```

computer.h

La méthode `addExpression` permet de créer une nouvelle expression avec une valeur de type **`int`** transmise en argument. Pour créer un nouvel objet `Expression`, la méthode *alloue dynamiquement* un objet `Expression` dont l'adresse est sauvegardée dans un tableau de pointeurs d'objets `Expression` qui a été alloué dynamiquement et dont l'adresse est stockée dans l'attribut `exps` de type `Expression**`. L'attribut `nb` représente le nombre d'adresses sauvegardées dans ce tableau. L'attribut `nbMax` représente le nombre maximum d'adresses qui peuvent être sauvegardées avant un agrandissement du tableau (*c.-à-d.* la taille du tableau pointé par `exps`). La méthode `agrandissementCapacite()` gère les éventuels besoins en agrandissement du tableau. La classe

`ExpressionManager` possède pour l'instant un unique constructeur sans argument. Initialement, le tableau pointé par `exps` ne contient aucune adresse.
Définir les méthodes de la classe `ExpressionManager`.

Question 5

La classe `ExpressionManager` nécessite t-elle le développement d'un destructeur ? Pourquoi ? Si oui, implémenter ce destructeur.

Question 6

Dans l'hypothèse où la duplication d'un objet `ExpressionManager` est autorisée, la classe `ExpressionManager` nécessite t-elle le développement d'un constructeur de recopie et/ou d'un opérateur d'affectation ? Si oui, implémenter ces méthodes.

Question 7

Définir la classe `Item` ainsi que l'ensemble de ses méthodes.

Question 8

La classe `Item` nécessite t-elle le développement d'un destructeur ? d'un constructeur de recopie ? d'un opérateur d'affectation ?

Question 9

Est-il possible de définir un tableau (alloué dynamiquement ou non) d'objets `Item` ? Expliquer. Est-il possible de créer un tableau (alloué dynamiquement ou non) de pointeurs d'objet `Item` ? Expliquer.

Question 10

Soit la classe `Pile` dont voici une définition partielle :

```
class Pile {
 Item* items;
 size_t nb;
 size_t nbMax;
 size_t nbAffiche;
 string message;
 void agrandissementCapacite();
public:
 Pile();
 void affiche() const;
 void push(Expression& e);
 void pop();
 bool estVide() const;
 size_t taille() const;
 Expression& top() const;
 void setMessage(const string& m);
 string getMessage() const;
 void setNbItemsToAffiche(size_t n);
};
```

computer.h

La méthode `affiche()` permet d'afficher un message à destination de l'utilisateur ainsi que l'état courant de la pile dans l'ordre où les objets `Item` ont été empilés. Le message utilisateur est contenu dans l'attribut `message` de type `string` et est modifiable avec la méthode `setMessage()`. La méthode `setNbItemsToAffiche()` permet de modifier le nombre d'items de la pile à afficher au maximum qui est stocké dans l'attribut `nbAffiche`. La méthode `push` permet d'empiler une nouvelle expression transmise par référence. L'adresse de cet objet `Expression` est encapsulée dans un objet `Item`. Pour cela, un tableau de pointeurs d'objets `Item` a été alloué dynamiquement. L'adresse de ce tableau est stockée dans l'attribut `items` de type `Item*`. L'attribut `nb` représente le nombre d'objets `Item` à considérer dans ce tableau (ceux qui sont considérés comme faisant partie de la pile). L'attribut `nbMax` représente la taille réelle du tableau, *c.-à-d.* le nombre d'expressions qui peuvent être empilées avant un agrandissement du tableau (*c.-à-d.* la taille du tableau pointé par `items`). La méthode `agrandissementCapacite()` gère les éventuels besoins en agrandissement du tableau. La méthode `top()` permet d'obtenir une référence sur l'expression au sommet de la pile, *c.-à-d.* le dernier élément empilé. La méthode `pop()` permet de dépiler l'objet `Item` au sommet de la pile.

La classe `Pile` possède pour l'instant un unique constructeur sans argument. Initialement, il n'y aucune expression dans la pile. Définir les méthodes de la classe `Pile`.

Question 11

Dans l'hypothèse où la duplication d'un objet `Pile` est autorisée, la classe `Pile` nécessite t-elle le développement d'un destructeur, d'un constructeur de recopie, et/ou d'un opérateur d'affectation ? Si oui, se contenter d'implémenter le destructeur.

Question 12

Définir les méthodes de la classe `Controleur` dont voici l'interface :

```
class Controleur {
 ExpressionManager& expMng;
 Pile& expAff;
public:
 Controleur(ExpressionManager& m, Pile& v):expMng(m), expAff(v){}
 void commande(const string& c);
 void executer();
};
```

computer.h

L'application devra être utilisable avec la fonction principale suivante :

```
/*...*/
#include "computer.h"

int main(){
 ExpressionManager expMng;
 Pile expAff;
 Controleur controleur(expMng,expAff);
 expAff.setMessage("Bienvenue");
 controleur.executer();
 return 0;
}
```

main.cpp