TP 4

1 L'art de l'Expression : Hiérarchie de classes Expression Arithmétique entière

On distingue 2 types d'expressions arithmétiques entières :
— les expressions correspondant à une constante entière (ex : 3)
— les expressions arithmétiques construites à partir d'un opérateur binaire et de 2 expressions (ex : exp1 * exp2)

Ecrivez une classe abstraite *Expression* qui offre la possibilité d'évaluer une expression arithmétique de manière transparente, c'est à dire que la fonction d'évaluation s'utilise sans avoir à connaître la nature exacte de l'expression. Cette classe devra comporter les deux méthodes `virtual int eval() const` et `virtual Expression * clone() const`, toutes deux virtuelles pure.

Vous développerez ensuite une hiérarchie de classes permettant de prendre en compte les différentes formes possibles d'une expression et correspondant à une gestion saine de la mémoire. Les différentes formes d'expressions correspondront à des structures de données différentes. Vous coderez au moins les classes suivantes : `Constante`, `Plus`, `Moins`, `Mult`.

La classe `Constante` aura un constructeur prenant un `int` en paramètre (la valeur de la constante entière). Les classes `Plus`, `Moins`, `Mult` stockeront des pointeurs vers des copies (clones) des expressions passées en paramètre lors de l'initialisation par le constructeur. Notez bien que l'on ne connaît pas le type exact des expressions à cloner et c'est la raison pour laquelle la hiérarchie de classe *Expression* offre une méthode de clonage. Nous avons en effet vu en cours qu'on ne peut pas recourir au constructeur par copie d'une expression pour la cloner avec toute sa spécificité. En plus de la spécialisation de la méthode `eval` qui est centrale à cet exercice, toutes les classes définiront donc aussi une spécialisation de la méthode `clone` qui duplique l'expression courante (en recourant à de l'allocation dynamique) et retourne un pointeur vers la copie.

Pour réaliser cet exercice, je vous recommande dans un premier temps de bloquer le mécanisme de copie dans vos classes (constructeur par copie, affectation, copie par déplacement, affectation par déplacement).

Pour ceux qui veulent aller un peu plus loin, il est possible (et même préférable) d'améliorer la hiérarchie de classe en codant une classe `ExpressionBinaire` qui va factoriser du code pour l'ensemble des expressions binaires.

On testera la classe obtenue avec un programme utilisateur du type :

```
int main()
{
  int a=5;
  const Expression & e = Mult(Plus( Constante(a), Constante(-2)),
                              Plus( Constante(1),
                                    Constante(3)) );
  std::cout << e.eval() << std::endl;
  return 0;
}
```

Pourquoi l'usage de la référence dans ce programme utilisateur est-il important ? Pouvez-vous appréhender le nombre de copies de chacunes des constantes lors de l'exécution de ce programme ? Posez-vous la question de l'espace mémoire engagé pour la création de votre `Expression` finale.

2 Objet Fonction

Avez-vous bien fait l'exercice du TP précédent portant sur les Objets Fonctions ?