TP 3

1 Construction d'une classe d'objets fonctions

Construisez une classe de générateurs d'entiers, multiples d'une valeur donnée. Les valeurs fournies par un générateur vont en augmentant au fil des sollicitations. Par exemple le générateur des multiples de 5 renverra les valeurs 5, puis 10, puis 15, etc.

2 Hiérarchie de classes Expression Arithmétique

On distingue 2 types d'expressions arithmétiques :
— 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 indépendamment de sa structure interne. 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 un pointeur vers une copie (un clone) 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 on ne peut pas recourir au constructeur par copie d'une expression pour la cloner avec toute sa spécificté. 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 réalisant une allocation dynamique) et retourne un pointeur vers la copie.

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 ?

3 De C++ à Java ou Vice Versa (le retour)

La correction fournie à cet exercice du TP1 fait apparaître deux implémentation `C++` de la classe `Image`. L'une d'entre elles est efficace et économique en mémoire (`Image`), l'autre non (`Image2`) ! Imaginons que l'on veuille que les images stockent des `Pixel`s polymorphes (avec deux spécialisations possibles suivant qu'il s'agit d'un Pixel en couleur ou en niveau de gris). Dans ce cas, il faudra que `Pixel` devienne la racine d'une hiérarchie de classes avec des fonctions membres virtuelles. Il faudra également modifier `Image2` de manière à ce qu'il y ait deux initialisations possibles suivant que les pointeurs sur `Pixels` pointent sur des `PixelsCouleurs` ou des `PixelsNiveauDeGris`. Il n'y a donc pas une hiérarchie de classe `Image`, mais des `Pixel`s de nature différente suivant le constructeur appelé. Faites les modifications requises. Que constatez-vous en terme de surcoût en temps d'exécution et de mémoire ? Le langage `C++` est-il encore compétitif par rapport à `Java` ?