TP 2

1 Manipulation du qualificatif *const*

```
double d;
const double r;
const double pi = 3.1416;
double *ptr = &pi;
double *const cpt;
double *const ptd = &d;
const double *ctd = &d;
const double *ptc = &pi;
double *const ptp = &pi;
const double *const ppi = &pi;
double *const *pptr1 = &ptc;
double *const *pptr2 = &ptd;

void F ()
{
  ptr = new double;
  r = 1.0;
  *ptr = 2.0;
  cpt = new double;
  *cpt = 3.0;
  ptc = new double;
  *ptc = 4.0;
  ptd = new double;
  *ptd = 5.0;
  ctd = new double;
  *ctd = 6.0;
  ptp = new double;
  *ptp = 7.0;
  ppi = new double;
  *ppi = 8.0;
}
```

Indiquez quelles déclarations et instructions sont légales ou pas. Compilez-les pour vérifier vos réponses.

Utilisez l'opérateur de conversion dédié pour duper le compilateur et vérifiez que le comportement "attendu" du programme n'est plus garanti dans ce cas.

2 Expressions arithmétiques et polymorphisme

On distingue 3 types d'expressions arithmétiques :
- les expressions correspondant à une constante (par exemple `3`)
- les expressions construites à partir d'un opérateur unaire et d'une expression (par exemple `-exp`)
- les expressions arithmétiques construites à partir d'un opérateur binaire et de deux expressions (par exemple `exp1 * exp2`)

Écrivez une classe abstraite `Expression` qui offre la possibilité d'évaluer une expression arithmétique indépendamment de sa structure interne. Cette classe comportera au moins 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 possibles d'expressions correspondront à des structures de données différentes. Vous coderez au moins les classes suivantes : `Constante`, `MoinsUnaire`, `Plus`, `Moins`, `Mult`.

La classe `Constante` aura un constructeur prenant un `int` en paramètre (la valeur de la constante). La classe `MoinsUnaire` prendra en paramètre une référence constante vers une expression, les trois autres classes prendront en paramètre deux références constantes vers deux expressions. Toutes ces classes définirons la méthode `eval` qui est centrale à cet exercice, ainsi que la méthode `clone` qui duplique l'expression courante (en réalisant une allocation dynamique) et retourne un pointeur vers la copie.

Les quatre classes `MoinsUnaire`, `Plus`, `Moins`, `Mult` stockerons un pointeur vers une copie de la/les expression(s) passée(s) en paramètre. Il est important de stocker une copie (donc obtenue par un appel à la méthode `clone`) sous réserve d'avoir un problème lors de l'évaluation.

Pour ceux qui veulent aller un peu plus loin, il est possible (et même préférable) d'améliorer la hiérarchie de classe en codant les classes `ExpressionUnaire` et `ExpressionBinaire` qui vont factoriser du code et des comportements de toutes les expressions unaires et binaires.

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

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

3 Opérateur d'affectation entre objets polymorphes

On considère la classe polymorphe Produit dont héritent les classes ProduitFrais et ProduitLuxe. La classe ProduitLuxeHorsTaxe hérite de la classe ProduitLuxe. Écrire un opérateur d'affectation pour la classe Produit qui s'adapte au type dynamique de ses deux opérandes.

On reprendra, en les comprenant, et en les étendant, les deux solutions possibles présentées en cours.