TP 8

1 Métaprogrammation et Listes de Types

Utilisez les template du C++ pour mettre en œuvre le concept de liste de types, ainsi que les principales fonctionnalités vues en cours (longueur d'une liste de type, type en ième postion, recherche d'un type dans une liste de type...).

2 Pointeurs intelligents

Pour compléter ce qui a été vu en cours, regarder les exemples d'utilisation des pointeurs intelligents (`unique_ptr`, `shared_ptr`, `weak_ptr`) sur (`https://www.cplusplus.com/reference/memory/`). Etes-vous prêts à coder vous même une classe `intrusive_ptr` ou `shared_ptr`, après avoir relu soigneusement les slides du cours ? Dans un premier temps, vous pouvez écarter la possibilité de gérer le polymorphisme.

3 Manipulation du qualificatif *const*

Cet exercice vous éclairera sur les dangers d'utilisation de *const_cast*. Il revient au programmeur de mettre en place des librairies pour lesquelles l'utilisateur n'ait pas à utiliser cet opérateur de conversion. Cela signifie que dans votre activité de programmeur vous devrez toujours veiller à mettre const partout où ca sera nécessaire.

```
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 tenter de duper le compilateur et vérifiez que le comportement "attendu" du programme n'est plus garanti dans ce cas.

4 Les Visiteurs

Etant données une hiérarchie de classe `Figure` et une hiérarchie de classe `Visiteur`, on souhaite établir une fonctionnalité de visite d'une `Figure` par un `Visiteur` qui soit polymorphe vis à vis du type dynamique du `Visiteur` mais aussi vis à vis du type dynamique de la `Figure`.

Cet exercice est difficile et par conséquent optionnel. Une alternative est de commencer par revenir sur l'exemple de l'affectation entre Produits vu dans le cours sur le polymorphisme (dans cet exemple on présuppose qu'on a affaire à une application où l'affectation d'un type de Produit à un autre a un sens), qui permettait également d'appréhender la difficulté de la gestion du polymorphisme sur plusieurs arguments. Dans ce cas là également, il pourrait être utile de recourir à la métaprogrammation !

4.1 Hiérarchie de classes Figure

Ecrire une hiérarchie de classes `Figure` munie de 2 fonctions membre d'affichage `affichageContour()` et `affichagePlein()`. Il pourra par exemple y avoir les classes `Carre` et `Rond`, et les deux fonctions d'affichage se contenteront ici d'écrire du texte à l'écran. Ces fonctions pourront ne pas être virtuelles. En revanche, la hiérarchie de classe `Figure` devra être munie d'une fonction membre virtuelle `accepte` activant le passage de `Visiteur`s pour affichage selon un mode spécifique qui pourra par exemple combiner les 2 affichages nativement présents dans les `Figure`.

4.2 Hiérarchie de classes Visiteur

On souhaite à présent mettre en place la hiérarchie de classe `Visiteur`. Les objets de cette hiérarchie permettront d'afficher des figures selon un mode qui sera propre à leur spécialisation. **L'entrée d'une nouvelle classe dans cette hiérarchie ne devra pas entraîner de modifications dans la hiérarchie de classes `Figures`.**

Dans le programme principal, on manipulera des `Figure`s et des `Visiteur`s sans connaître leur type exact par le biais de références ou de pointeurs. Lorsque qu'une `Figure` référencée par `f` doit être affichée par un `Visiteur` référencé par `v`, cela est réalisé par appel à `f.accepte(v)`. Le polymorphisme vis à vis de `f` est assuré par le fait que `f.accepte` est virtuelle et pourra s'adapter au fait que `f` est un `Carre` ou un `Rond`. Dans l'appel spécialisé à `f.accepte`, le polymorphisme par rapport à `v` sera réalisé par appel à une fonction membre virtuelle `v.visiteCarre` qui s'adaptera au type exact de `v` .

Comme étudié en cours, toutes les classes de `Visiteurs` doivent donc être dotées des fonctions membres virtuelles `visiteCarre()` et `visiteRond()`. En bénéficiant du mécanisme de surcharge, ces fonctions peuvent toutes s'appeler `visite` puisqu'elles s'appliquent à des paramètres de type différent.

Si on connaît la liste des types correspondant aux spécialisations possibles de `Figure`, ces fonctions membres `visite` peuvent être générées automatiquement dans toutes les classes de la hiérarchie `Visiteur`, en utilisant la métaprogrammation.

On a pour cela besoin d'un patron de visiteur générique quant au type d'objet qu'il visite, et d'un patron de visiteur multiple qui sera à la fois un visiteur pour chacun des types d'une liste de type donnée (cf. transparents du cours où l'on montre que l'on utilise pour cela de l'héritage multiple).

```
template<class FF>
class PatronVisiteur;

template<class TList, template<class> class PVisiteur> // TList étant une liste de types
class VisiteurMultiple;
```

Attention de ne pas oublier d'utiliser un mécanisme similaire pour fabriquer la classe de base `Visiteur` (utilisation d'un template et de l'héritage pour doter la classe de toutes les signatures de fonctions membres virtuelles pures `visite()`.

```
template<class FF>
class PatronBaseVisiteur;

template<class TList>
class BaseVisiteurMultiple;
```

Donnez les spécialisations nécessaires pour rendre votre classe de base et vos spécialisations de visiteurs opérationnelles dans un petit programme de test manipulant des objets et des visiteurs à travers des pointeurs ou des références.

```
int main(int argc, char **argv)
{
  Figure * pf1=new Carre;
  Figure * pf2=new Rond;
  Visiteur * pv=new VisiteurAffichagePlein;
  pf1->accepte(pv);
  pf2->accepte(pv);
  delete pf1;
  delete pf2;
  return 0;
}
```

Remarque :

C++17 s'est doté d'un mécanisme de visiteur où le visiteur doit être une fonction *(callable)* au sens large du terme (fonction, lambda fonction ou objet fonction). Il s'agit d'un mécanisme différent du Design Pattern Visitor vu plus haut. Vous pouvez étudier l'exemple de la documentation fournie par cpp.reference.com.