TP de C++ sur la programmation par extensions

Le but de ce TP est d'utiliser la méthode de programmation par extension au travers d'un petit exemple. La programmation fera intervenir les notions de classes, **classes abstraites, classes dérivées, héritage multiple**. Ses notions vous permettront de mettre en œuvre les concepts de **polymorphisme** et d' **aiguillage dynamique**. Nous utiliserons de plus la notion de **variables de classe**.

Nous voulons concevoir et programmer des classes nous permettant de manipuler un certains nombre de figures fermées. Nous tiendrons simplement compte de certaines propriétés de ces figures de sorte que nous implémenterons seulement certaines fonctions :

— chaque figure doit pouvoir donner le calcul de son périmétre ;

— chaque figure doit être capable d'afficher ses caractéristiques (pour un cercle par exemple, c'est la longueur de son rayon, pour un carré, la longueur de son côté, etc ...).

Exercice 1

1. Définir et implémenter une classe `Figure` qui sera un parent commun pour tous les types de figure que l'on souhaite pouvoir définir. Créer les deux fonctions (`perimetre()` et `afficherCaracteristiques()`) correspondant aux propriétés énoncées ci-dessus. On ne peut définir encore le corps de ces 2 fonctions. Ce sont des fonctions **virtuelles pures** aussi appelées **abstraites** et la classe **Figure** devient donc aussi abstraite. Une fonction virtuelle pure se déclare ainsi : `virtual int perimetre()=0`.
2. On souhaite ensuite pouvoir manipuler des polygones (qui sont un cas particulier de figures fermées). Définir la classe `Polygone` contenant simplement un attribut `nbCotes`. Ecrire les fonctions utiles :
   — le constructeur ;
   — une version de la fonction `afficherCaracteristiques()`.

   En revanche, est-ce utile de définir dès à présent le calcul du périmètre ?
3. On souhaite maintenant manipuler des rectangles et des carrés (**que l'on ne souhaite pas positionner à des coordonnées précises dans un plan ; on veut juste les caractériser par la longueur de leur côtés**). Définir la classe `Rectangle` (qui est un polygone particulier) et l'ensemble des fonctions utiles :
   — le constructeur ;
   — des fonctions qui permettent de consulter et de modifier la longueur des côtés ;
   — la fonction `perimetre()` ;
   — une nouvelle version d'`afficherCaracteristiques()` (que faites-vous de la version de cette fonction définie dans la classe `Polygone` ?).

Définir ensuite de façon très courte la classe `Carre`.

4. On souhaite manipuler des triangles (équilatéraux seulement). De même que pour le rectangle, définir la classe `TriangleEquilateral` (qui est un polygone) ainsi que les fonctions utiles.
5. On souhaite maintenant manipuler des cercles (qui sont un autre exemple de figures fermées). Définir la classe `Cercle` ainsi que les fonctions utiles :
   — le constructeur ;
   — des fonctions qui permettent de consulter et de modifier la longueur du rayon ;
   — la fonction `perimetre()` ;
   — une nouvelle version d'`afficherCaracteristiques()`.
6. Ecrire un programme de test qui fera les choses suivantes :
   — création d'un tableau de figures fermées ;
   — initialisation de ce tableau qui doit contenir au moins un carré, un cercle et un triangle equilatéral ;
   — parcours du tableau et affichage pour chaque figure de son périmètre et de ses caractéristiques.

   Ce programme principal vous permet de mettre en évidence les notions de **polymorphisme** et de **liaison dynamique**.
7. **Variables de classe** : on veut maintenant être capable de compter le nombre de figures créées, sans pour cela avoir à les compter dans le programme de test. Pour cela, un des moyens est d'utiliser une variable de classe. Placez cette variable et faites en sorte de l'utiliser pour compter le nombre de figures créées (sans rien toucher au programme de test).
8. **Héritage multiple** : soit la classe C++ suivante (pensez à inclure string) :

```
class Coloriable
{
 protected :
  string couleur;

 public :
  Coloriable(string coul){
    couleur=coul;

  }

  string getCouleur(){
    return couleur;
  }

  void setCouleur(string coul){
    couleur=coul;
  }
};
```

— inclure cette classe dans votre fichier (il vous faut l'écrire !)

— faire ensuite en sorte que chaque cercle soit coloriable par **héritage multiple** (pensez à modifier le constructeur) ;

— modifier le programme principal de façon à positionner la couleur de chaque cercle créé lors de l'initialisation de votre tableau de figures. (Pour le moment, nous ne sommes pas encore capables lors de la consultation des figures du tableau de savoir si une figure est un cercle et donc d'appeler la méthode `getCouleur` pour cet objet ; nous le ferons lors de l'exercice 2).

Exercice 2 : version longue

En fonction de votre avancement dans le TP (à discuter avec votre enseignant de TP) choisissez soit la version longue de l'exercice 2, soit la version courte.

On désire maintenant mettre des figures dans une file. On va donc créer tout d'abord des classes permettant d'implémenter en C++ une file générique.

1. Créer une classe `Cellule` qui contient de quoi implémenter une structure chainée générique. On protégera bien sûr les attributs et on créera simplement un constructeur.
2. Créer une classe `File` qui utilise la classe `Cellule` (une File est par exemple composée de l'adresse de la première cellule, de l'adresse de la dernière cellule et du nombre d'éléments). De façon à simplifier les accès aux attributs de `Cellule` dans la classe `File`, on déclarera la classe `File` amie de la classe `Cellule`. Ecrire le constructeur et les opérations permettant :
   — d'insérer un élément
   — de supprimer un élément
   — de savoir si la file est vide
   — de récupérer la valeur de l'élément de tete (resp. de queue)
   — de récupérer le nombre d'éléments de la file
3. Créer de plus une classe `Vide` interne à la classe `File` qui sert d'exception lorsque on effectue une opération illicite sur une file vide.
4. Créer ensuite un programme principal qui manipule cette File en l'instanciant avec des pointeurs de Figures. Enfiler des figures de différents types (carrés, cercles, rectangles et triangles). Puis, en utilisant le polymorphisme et l'aiguillage dynamique, défiler chaque figure et appeler l'opération `perimetre` et `afficherCaracteristiques` sur chaque figure.
5. Lorsque la figure est une instance de `Cercle`, vous afficherez aussi sa couleur. Pour cela, il vous faudra utiliser la fonction `typeid` qui fonctionne de la façon suivante :

```
Figure * f = new Cercle(6,"rouge");

if (typeid(*f)==typeid(Cercle)) {
  // afficher la couleur
}
```

Il faut ajouter en en-tête de votre fichier :

```
#include <typeinfo>
```

Exercice 2 : version courte

On désire maintenant mettre des figures dans une file en s'appuyant sur la bibliothèque STL.

1. Instancier la classe générique list issue de la STL. Pour cela, il faut inclure la description des listes avec l'instruction :

   ```
   #include <list>
   ```

2. Utiliser les méthodes et objets suivant vous permettant :
   - d'insérer un élément : push_back(un élément)
   - de supprimer un élément : pop_back()
   - de savoir si la liste est vide : empty()
   - de créer un itérateur pour parcourir la liste :

     ```
     list<Figure*>::iterator it;
     ```

   - de renvoyer un itérateur sur le premier élément de la liste : begin()
   - de tester si l'itérateur est arrivé à la fin de la liste : end()
3. Créer ensuite un programme principal qui manipule cette liste en l'instanciant avec des pointeurs de Figures. Enfiler des figures de différents types (carrés, cercles, rectangles et triangles). Puis, en utilisant le polymorphisme et l'aiguillage dynamique, défiler chaque figure et appeler l'opération `perimetre` et `afficherCaracteristiques` sur chaque figure.
4. Lorsque la figure est une instance de `Cercle`, vous afficherez aussi sa couleur. Pour cela, il vous faudra utiliser la fonction `typeid` qui fonctionne de la façon suivante :

   ```
   Figure * f = new Cercle(6,"rouge");

   if (typeid(*f)==typeid(Cercle)) {
     // afficher la couleur
   }
   ```

   Il faut ajouter en en-tête de votre fichier :

   ```
   #include <typeinfo>
   ```