TP Semaine 8 (2 séances)
Hiérarchie Formes géométriques

Le but de ce TP est de compléter puis de manipuler par polymorphisme une hiérarchie d'héritage de taille conséquente en C++. Nous nous appuierons sur un exemple de hiérarchie de classes de formes géométriques introduite en TD.

1 La hiérarchie de classes de formes

Une partie de cette hiérarchie est déjà implémentée : récupérez chez vous le répertoire (et la totalité de son contenu) `/net/Bibliotheque/AP2/TP_par_Semaine/Semaine08`.

Etude de la hiérarchie et du code C++ fourni

Exercice 25 : Le but ici est de se familiariser avec l'ensemble du code fourni. Pour cela :

1. Consultez le fichier `A_LIRE.txt` qui vous servira de guide.
2. Consultez le code de l'ensemble des fichiers des classes des formes, et dessinez sur votre feuille la hiérarchie d'héritage mise en oeuvre.
3. Compilez, testez, lisez le reste du code, etc... En particulier, soyez sûrs d'avoir bien compris le rôle de la classe `Screen` et de la manière de l'utiliser dans les autres classes et dans le `main`.

Rajout de classes dans la hiérarchie

On souhaite compléter la hiérarchie avec 2 classes de formes : `Triangle` qui hérite de la classe abstraite `Shape`, et `Square` qui hérite de `Rectangle`.

Exercice 26 : Implémentez ces deux nouvelles classes :

1. complétez les fichiers vides à votre disposition, `Triangle.*` et `Square.*`,
2. et testez les en décommentant leur utilisation dans le `main`.
3. Les fichiers `util.*` contiennent des fonctions de saisie des formes qui seront utiles pour la suite. Rajoutez les fonctions de saisie pour les deux nouvelles formes que vous venez d'implémenter.

2 Gestion d'un dessin = ensemble de formes

Avant de commencer cette partie, recopiez dans votre répertoire de travail sur les formes les fichiers contenus dans le sous-répertoire `Supplements` :

- `DessinShapes.*` qui sont vides,
- `main.cc` à décommenter et à compléter,
- `Makefile` complet.

Vous trouverez également dans ce sous-répertoire `Supplements` un exécutable de la version finale du programme que vous allez développer au cours de ce TP. Testez le pour vous faire une idée précise des fonctionnalités attendues.

On souhaite donc écrire ici un programme permettant de créer et de manipuler un ensemble de formes dessinées à l'écran, en interaction avec un utilisateur via un menu.

Pour cela, il faut une structure de données qui permet de stocker les formes créées pour pouvoir à tout moment les afficher, les modifier, etc... Nous proposons d'utiliser un **tableau de pointeurs sur `Shape`**. Ce tableau sera mis à jour à chaque ajout/suppression de formes, et sera parcouru "par polymorphisme" pour l'affichage ou toute modification des formes.

La classe `DessinShapes`

Pour rendre le code plus lisible et plus modulaire, on propose de gérer ce tableau de pointeurs sur `Shape` au sein d'une classe dont voici une entête (minimum) :

```
class DessinShapes
{
  private:
    static const int MAX = 100;
    Shape * my_tabShapes[MAX];
    int my_nbShapes;
  public:
    DessinShapes();
    ~DessinShapes();
    void addShape( Shape * pshape );
    void refresh( Screen & s );
};
```

Les attributs `my_tabShapes` et `my_nbShapes` servent évidemment à stocker les pointeurs sur les formes créées par ailleurs (dans le `main` par exemple). La méthode `addShape` sert à rajouter un pointeur au tableau, et la méthode `refresh` à afficher l'ensemble des formes à l'écran. Voici un exemple d'utilisation classique de ces méthodes si la variable `dessin` est de type `DessinShapes` :

```
dessin.addShape( new Line(black, Point(4,5), Point(15,22)) );
dessin.refresh( ecran );
```

Exercice 27 : Implémentez cette version minimum de la classe `DessinShapes`. Cela vous permettra ainsi d'obtenir les fonctionnalités 1 à 6 du menu proposées dans le `main`. Implémentez une à une ces fonctionnalités d'ajout des 6 formes et de leur affichage à l'écran. *Remarque :* pensez que les fichiers `util.*` contiennent des fonctions de saisie des paramètres pour les formes, utilisez les !

3 Rajout de fonctionnalités

S'appliquant à toutes les formes (c, d et e dans menu)

Exercice 28 : Rajoutez à la classe `DessinShapes` les 3 méthodes suivantes :

```
void setColourAll( char col );
void moveAll( int dx, int dy );
void eraseAll();
```

et utilisez les pour implémenter les options c, d et e du menu.

S'appliquant seulement à une forme (x, y et z dans menu)

Ces fonctionnalités se réalisent en deux étapes :

1. sélection d'une forme,
2. application de la modification sur cette forme.

L'étape de sélection d'une forme peut être réalisée de plusieurs faÃ§ons différentes, par exemple en fonction de l'ordre de création des formes (indice dans le tableau), ou selon l'appartenance ou non d'un point à cette forme.

Exercice 29 : Avec une sélection par indice.
Rajoutez à la classe `DessinShapes` la méthode `Shape * select( int ind ) const` qui retourne le pointeur sur la forme d'indice `ind`, ou `NULL` si l'indice est mauvais. Il suffit alors d'appliquer la bonne fonction sur le pointeur de forme obtenu pour implémenter les options x, y et z du menu.
Remarque : il y a une subtilité pour l'option z, il faut aussi modifier le tableau de formes... donc, une nouvelle fonction dans la classe `DessinShapes` serait bien utile pour effacer "proprement" la forme d'indice `ind` du dessin : `void erase( int ind )` .

Exercice 30 : Avec une sélection par point contenu dans la forme.
Même chose que précédemment, mais en utilisant comme fonction de sélection une méthode qui retourne le pointeur sur la première forme du dessin contenant un point p, ou `NULL` sinon : `Shape * select( const Point & p ) const`
Cela implique donc de rajouter dans la hiérachie des formes une méthode qui permet de tester l'appartenance d'un point à une forme. Pour certaines formes, le calcul géométrique exact peut s'avérer très compliqué... Vous vous restreindrez à tester si le point appartient à la plus petite boite englobant la forme.

4 Pour aller plus loin

Modifier le rayon d'un cercle (r dans menu)

On souhaite offrir la possibilité à l'utilisateur de sélectionner un cercle du dessin (avec une des méthodes vues avant), puis lui appliquer une méthode de modification de la valeur du rayon (`void setRadius( int r )` par exemple). Cette méthode `setRadius` n'a bien sÃ»r un sens QUE pour la classe `Circle`, toutes les autres classes de la hiérarchie n'ont pas à contenir cette méthode.
Voici donc un bon exemple pour vous essayer au transtypage !

Dessin d'un cercle

Dans la classe `Screen`, la méthode `put_circle` qui permet de dessiner un cercle discret est vide. Implémentez la grâce à l'algorithme fourni en Annexe ci-après.

Annexe. L'algorithme de dessin d'un cercle discret de Bresenham.

C'est un algorithme classique, largement commenté et étudié dans tous les bons livres sur le graphisme.

L'algorithme décrit ci-dessous dessine un cercle dont le centre a pour coordonnées $(0, 0)$. Vous l'adapterez à votre cas.

Principe. À chaque tour de boucle, on calcule les coordonnées d'un point du cercle dans un octant du plan, et par symétrie, on obtient les 7 autres. Pour choisir quels points à coordonnées entières "donnent la meilleure approximation" des points réels du cercle, on gère le calcul d'une erreur sur la distance (d) entre ces points.

```
// Affiche les points dans chacun des 8 octants
Action affiche_points_cercle (point pt)
Debut
  Affiche(pt.x, pt.y)   ; Affiche(pt.y, pt.x)
  Affiche(pt.y, -pt.x)  ; Affiche(pt.x, -pt.y)
  Affiche(-pt.x, -pt.y) ; Affiche(-pt.y, -pt.x)
  Affiche(-pt.y, pt.x)  ; Affiche(-pt.x, pt.y)
Fin
// Affiche un cercle grace a l'algorithme de Bresenham
Action dessine_cercle()
Debut
  point pt(0, radius);
  int d = 3 - 2 * radius;
  Tant Que (pt.x < pt.y)
  Faire Debut
           affiche_points_cercle(pt)
           Si (d < 0)
           Alors   d += 4 * pt.x + 6;
           Sinon Debut
```

```
                d += 4 * (pt.x - pt.y) + 10;
                pt.y--;
              Fin
          pt.x++;
        Fin
  Si (pt.x == pt.y)
  Alors  affiche_points_cercle(pt)
Fin
```