Série N°4 TD/TP
Module : Programmation Orientée Objet en C++
Filière : Génie Informatique
Niveau : GI1, AU : 2023/2024

Le but de cet exercice est de vous illustrer les problèmes qui peuvent se poser lorsque l'on veut manipuler des objets de façon polymorphique (un objet pouvant se substituer à un autre objet).

Il s'agit de manipuler quelques formes géométriques simples en leur associant une méthode qui affiche leur description.

- Définissez une classe **Forme** en la dotant d'une méthode **void description()** qui affiche à l'écran : « Ceci est une forme ! » .
- Ajoutez au programme une classe **Cercle** héritant de la classe **Forme**, et possédant la méthode **void description()** qui affiche à l'écran : « Ceci est un cercle. ».
- Recopiez ensuite la fonction **main()**suivante, et testez votre programme :
- Ajoutez maintenant ceci à la fin de la fonction **main():**

```
int main() {
 Forme f;
 Cercle c;
f.description();
 c.description();
 return 0;
}
```

Forme f2(c); f2.description();

- Testez ensuite à nouveau votre programme.
- Voyez-vous vu la nuance ? Pourquoi a-t-on ce fonctionnement ?

Continuons nos prospections...

- Ajoutez encore au programme une fonction :

void affichageDesc(Forme& f)

```
int main() {
 Forme f;
 Cercle c;
affichageDesc(f);
 affichageDesc(c);
 return 0;
}
```

qui affiche la description de la forme passée en argument en utilisant sa méthode **description().**

- Finalement, modifiez le main ainsi :
- Testez le programme... **Le résultat vous semble-t-il satisfaisant ?** Pourquoi?
- Modifiez le programme (ajoutez 1 seul mot) pour que le résultat soit plus conforme à ce que l'on pourrait attendre.

Formes abstraites

Dans un nouveau projet, modifiez la classe ***Forme*** de manière à en faire une **classe abstraite** en lui ajoutant la méthode virtuelle pure **double *aire()***permettant de calculer l'aire d'une forme.

Écrivez également une classe **Triangle** et modifiez la classe **Cercle** existante héritant toutes deux de la classe **Forme**, et implémentant les méthodes **aire()** et **description().**

Vous l'aurez deviné, la classe **Triangle** aura comme attributs **base** et **hauteur** ainsi qu'un constructeur adéquat, et **Cercle** aura comme seul attribut rayon (ainsi qu'un constructeur adéquat).

Modifiez la fonction **affichageDesc** pour qu'elle affiche, en plus, l'aire de la forme passée en paramètre et testez avec la fonction main suivante :

```
int main() {

 Cercle c(5);

 Triangle t(10, 2);
affichageDesc(t);
 affichageDesc(c);
 return 0;
}
```

Classe Figure

Définissez les classes suivantes :

La classe abstraite **Figure**, possédant deux méthodes et aucun attribut :

- **affiche(),** méthode publique, constante, virtuelle pure, et ne prenant aucun argument.

  - une méthode **Figure* copie() const**, également virtuelle pure, chargée de faire une copie en mémoire de l'objet et de retourner le pointeur sur cette copie
- Trois sous-classes (héritage publique) de **Figure** : **Cercle**, **Carre** et **Triangle.**
- Une classe nommée **Dessin**, qui modélise une collection de figures. Il s'agira d'une collection «hétérogène» d'éléments créés dynamiquement par une méthode ad-hoc définie plus bas (**ajouteFigure**).

Pour chacune des classes **Cercle**, **Carre** et **Triangle**, définissez les attributs (privés/protégés) requis pour modéliser les objets correspondants.

Définissez également, pour chacune de ces sous-classes, un constructeur pouvant être utilisé comme constructeur par défaut, un constructeur de copie et un destructeur.

Dans les trois cas, affichez un message indiquant le type de l'objet et la nature du constructeur/destructeur.

Définissez la méthode de **copie** en utilisant le constructeur de **copie**.

Finalement, définissez la méthode **virtuelle affiche**, affichant le type de l'instance et la valeur de ses attributs.

Ajoutez un **destructeur** explicite pour la classe **Dessin**, et définissez-le de sorte qu'il détruise les figures stockées dans la collection en libérant leur espace mémoire (puisque c'est cette classeDessin qui, dans sa méthode **ajouteFigure**, alloue cette mémoire).

Comme pour les autres destructeurs, affichez en début de bloc un message, afin de permettre le suivi du déroulement des opérations.

Prototypez et définissez ensuite les méthodes suivantes à la classe **Dessin** :

- **void ajouteFigure(Figure const& fig)**

  qui ajoute à la collection une copie de la figure donnée en paramètre en faisant appel à sa méthode copie.
- **void affiche() const** qui affiche tous les éléments de la collection.

Testez votre programme avec le **main** suivant:

```
int main () {
  Dessin dessin;

  dessin.ajouteFigure(Triangle(3,4));
  dessin.ajouteFigure(Carre(2));
  dessin.ajouteFigure(Triangle(6,1));
  dessin.ajouteFigure(Cercle(12));

cout << endl << "Affichage du dessin : "
<< endl;
  dessin.affiche();
```

Compilez en ignorant pour l'instant les warnings. Testez le programme et assurez vous de bien comprendre l'origine de tous les messages affichés.

Si vous avez suivi les directives de l'énoncé, votre programme devrait, en dernier lieu, indiquer que le destructeur du dessin est invoqué... mais pas les destructeurs des figures stockées dans le dessin !

Pourquoi ?

Voyez-vous un moyen permettant de pallier cela ?

Corrigez (en ajoutant 1 seul mot) votre programme.

Ajoutez la fonction suivante, juste avant votre main :

```
void unCercleDePlus(Dessin const& img)
{
  Dessin tmp(img);
  tmp.ajouteFigure(Cercle(1));
  cout << "Affichage de 'tmp': " << endl;
  tmp.affiche();
}
```

Appelez cette fonction depuis main (n'importe où après la première instruction d'ajout de figure).

Si tout se passe « normalement », le système doit interrompre prématurément votre programme, en vous adressant un message hargneux

(genre «segmentation fault», ou «bus error»). (Il est cependant possible que sur certains systèmes cette erreur, qui se produit pourtant effectivement, ne soit pas détectée et ne donne lieu à aucun comportement visible.)

Quel est, à votre avis, le motif pour un tel comportement ? Corrigez votre programme.