TP 4 POO C++: Classes abstraites

IMT Atlantique – TAF Robin/Ascii

Programmation des systèmes embarqués

1 Polymorphisme et classes abstraites

Le but de cette partie 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).

Exercice 1 – Formes polymorphiques

Dans cet exercice, on souhaite manipuler des formes géométriques simples en leur associant une méthode qui affiche leur description.

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

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

4. Ajoutez maintenant ceci à la fin de la fonction `main()` :
   `Forme f2(c);`
   `f2.description();`
5. Testez ensuite à nouveau votre programme.
   — Voyez-vous vu la nuance ?
   — Pourquoi a-t-on ce fonctionnement ?
6. Ajoutez encore au programme une fonction : `void affichageDesc(Forme& f)`, qui affiche la description de la forme passée en argument en utilisant sa méthode `description()`. Modifiez le main ainsi :

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

7. Testez à nouveau votre programme. Le résultat vous semble-t-il satisfaisant ? Pourquoi ?
8. Modifiez le programme (ajoutez 1 seul mot) pour que le résultat soit plus conforme à ce que l'on pourrait attendre.

Exercice 2 – Formes abstraites

1. Recopiez le programme précédent dans le fichier `formesabstraites.cpp`, et 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.

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

3. 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);
  afficheDesc(t);
  afficheDesc(c);
  return 0;
}
```

Exercice 3 – Encore plus de formes

Le but de cet exercice est d'arriver à définir une collection hétérogène de figures géométriques.

1. Prototypez et définissez les classes suivantes : (vous pouvez adapter ce qui avait été réalisé dans l'exercice précédent)
   - La classe abstraite `Forme`, possédant deux méthodes et aucun attribut :
     - `affiche()`, méthode publique, constante, virtuelle pure, et ne prenant aucun argument.
     - une méthode `Forme* 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 `Forme` : `Cercle`, `Carre` et `Triangle`.
   - Une classe nommée `Dessin`, qui modélise une collection de formes. Il s'agira d'une collection «hétérogène» d'éléments créés dynamiquement par une méthode ad-hoc définie plus bas (`ajouteForme`).

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

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

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

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

6. Définissez la méthode virtuelle `affiche`, affichant le type de l'instance et la valeur de ses attributs.

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

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

9. Prototypez et définissez ensuite les méthodes suivantes à la classe `Dessin` :

   - `void ajouteForme(const Forme& f)`
     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 de tous les éléments de la collection.

10. Testez votre programme avec le main suivant :

```
int main()
{
  Dessin dessin;
  dessin.ajouteForme(Triangle(3,4));
  dessin.ajouteForme(Carre(2));
  dessin.ajouteForme(Triangle(6,1));
  dessin.ajouteForme(Cercle(2));
  cout << endl << "Affichage du dessin : " << endl;
  dessin.affiche();
  return 0;
}
```

    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 ?

11. Corrigez votre programme.

2 Renforcement

Exercice 4 – Jeu de stratégie

Dans cet exercice, on s'intéresse à modéliser un jeu de stratégie où des personnages de type dragons et hydres vont s'affronter. Un programme principal `dragons.cc` est fourni qui met en scène un combat entre un dragon et une hydre. La hiérarchie de classes qui permet de modéliser les créatures de ce jeu manque et il vous est demandé de la compléter.

La classe `Creature` : Une créature est caractérisée par :
- son nom (`nom_`, une chaîne de caractère constante) ;
- son niveau (`niveau_`, un entier) ;
- des points de vie (`points_de_vie_`, un entier) ;
- sa force (`force_`, un entier) ;
- et sa position (`position_`, encore un entier ; nous supposons que notre jeu est en 1D pour simplifier).

Ces attributs seront accessibles dans les classes dérivées de Creature.

Les méthodes à prévoir pour cette classe sont :

- un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force et la position de le créature au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;
- une méthode `bool est Vivant()` retournant `true` si la créature est vivante (nombre de points de vie supérieur à zéro) et `false` sinon ;
- une méthode `points_attaque` retournant les points d'attaque que la créature peut infliger ; il s'agit du niveau multiplié par la force si l'animal est vivant et zéro sinon ;

— une méthode `deplacer(int)`, ne retournant rien et ajoutant en entier passé en paramètre à la position de la créature ;

— une méthode `adieux()` ne retournant rien et affichant le message suivant : <nom> n'est plus ! <nom> est le nom de la créature ;

— une méthode `recevoirDegats`, ne retournant rien et retranchant au nombre de points de vie de la créature, si elle est vivante, un nombre de points passés en paramètre ; si la créature meurt, son nombre de points de vie sera mis à zéro et la méthode adieux invoquée ;

— une méthode `afficher()` ne retournant rien, affichant la créature en respectant le format suivant :
`<nom>, niveau: <niveau>, points de vie: <points>, force: <force>, points d'attaque: <attaque>, position: <position>`

La classe `Dragon` : Un `Dragon` est une `Creature`. Il a pour caractéristique spécifique la portée de sa flamme (`portee_flamme_`, un entier). Ses méthodes spécifiques sont :

— un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force, la portée de la flamme et la position du dragon au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;

— une méthode `voler(int pos)` ne retournant rien et permettant au dragon de se déplacer à la position `pos` ;

— une méthode lancer_flamme(Creature& bete) ne retournant rien et simulant une attaque le dragon sur une créature :

1. si le dragon est vivant, que la créature l'est aussi et qu'elle est à portée de sa flamme, alors le dragon inflige ses points d'attaque à la créature ; cette dernière faiblit du nombre de points d'attaque ; Le dragon faiblit aussi ; il perd d points de vie où d est la distance le séparant de la créature (plus il lance sa flamme loin et plus il s'affaiblit) ;

2. si au terme de ce combat, le dragon est toujours en vie, il augmente son niveau d'une unité s'il a vaincu la créature (qu'elle n'est plus en vie) ;

La créature est à portée de flamme du dragon si la distance qui les sépare est inférieure ou égale à la portée de la flamme (utilisez la fonction `distance` fournie).

La classe `Hydre` : Une `Hydre` est une créature. Elle a pour caractéristiques spécifiques la longueur de son cou (`longueur_cou_`, un entier) ainsi que la dose de poison qu'elle peut injecter par attaque (`dose_poison_`, un entier). Ses méthodes spécifiques sont :

— un constructeur permettant d'initialiser le nom, le niveau, les points de vie, la force, la longueur du cou, la dose de poison et la position de l'hydre au moyen de valeurs passées en paramètre, dans cet ordre ; le constructeur acceptera zéro comme valeur par défaut pour la position ;

— une méthode `empoisonne(Creature& bete)` ne retournant rien et simulant ce qui se passe lorsque l'hydre empoisonne une créature :

1. si l'hydre est vivante, que la créature l'est aussi et qu'elle est à portée du cou, alors l'hydre inflige des dommages à la créature ; cette dernière faiblit du nombre de points d'attaque de l'hydre augmentés de la dose de poison ;

2. si au terme de ce combat, la créature n'est plus en vie, l'hydre augmente son niveau d'une unité ;

La créature est à « portée de cou » de l'hydre si la distance qui les sépare est inférieure ou égale à la longueur du cou (utilisez la fonction distance fournie).

Nous supposerons l'existence d'une fonction `combat` qui prend en paramètre un dragon et une hydre et fait en sorte que :
— l'hydre empoisonne le dragon,
— puis le dragon souffle sur l'hydre.

Exemples de déroulement

L'exemple de déroulement ci-dessous correspond au programme principal fourni.

```
Dragon rouge, niveau: 2, points de vie: 10, force: 3,
points d'attaque: 6, position: 0
se prépare au combat avec :
Hydre maléfique, niveau: 2, points de vie: 10, force: 1,
points d'attaque: 2, position: 42

1er combat :
     les créatures ne sont pas à portée, donc ne peuvent pas s'attaquer.
Après le combat :
Dragon rouge, niveau: 2, points de vie: 10, force: 3
points d'attaque: 6, position: 0
Hydre maléfique, niveau: 2, points de vie: 10, force: 1,
points d'attaque: 2, position: 42


Le dragon vole à proximité de l'hydre :
Dragon rouge, niveau: 2, points de vie: 10, force: 3,
points d'attaque: 6, position: 41


L'hydre recule d'un pas :
Hydre maléfique, niveau: 2, points de vie: 10, force: 1,
points d'attaque: 2, position: 43


2e combat :
     + l'hydre inflige au dragon une attaque de 3 points
        [ niveau (2) * force (1) + poison (1) = 3 ] ;
     + le dragon inflige à l'hydre une attaque de 6 points
        [ niveau (2) * force (3) = 6 ] ;
     + pendant son attaque, le dragon perd 2 points de vie supplémentaires
        [ correspondant à la distance entre le dragon et l'hydre : 43 - 41 = 2 ].
Après le combat :
Dragon rouge, niveau: 2, points de vie: 5, force: 3,
points d'attaque: 6, position: 41
Hydre maléfique, niveau: 2, points de vie: 4,
force: 1, points d'attaque: 2, position: 43

Le dragon avance d'un pas :
Dragon rouge, niveau: 2, points de vie: 5, force: 3,
points d'attaque: 6, position: 42


3e combat :
     + l'hydre inflige au dragon une attaque de 3 points
        [ niveau (2) * force (1) + poison (1) = 3 ] ;
     + le dragon inflige à l'hydre une attaque de 6 points
        [ niveau (2) * force (3) = 6 ] ;
     + pendant son attaque, le dragon perd 1 points de vie supplémentaires
        [ correspondant à la distance entre le dragon et l'hydre : 43 - 42 = 1 ].
     + l'hydre est vaincue et le dragon monte au niveau 3.
Hydre maléfique n'est plus !

Après le combat :
Dragon rouge, niveau: 3, points de vie: 1, force: 3,
```

points d’attaque: 9, position: 42
Hydre maléfique, niveau: 2, points de vie: 0,
force: 1, points d’attaque: 0, position: 43

4e combat :
    Quand une créature est vaincue, rien ne se passe..
Après le combat :
Dragon rouge, niveau: 3, points de vie: 1, force: 3,
points d’attaque: 9, position: 42
Hydre maléfique, niveau: 2, points de vie: 0,
force: 1, points d’attaque: 0, position: 43