Travaux dirigés C++ n°4

Informatique

—IMAC 2e année—

Héritage et polymorphisme

Le but de ce TD est de comprendre l'intérêt et de mettre en place les mécanismes d'héritage et du polymorphisme.

Nous voulons représenter des figures géométriques. Commençons par les rectangles. Pour les représenter, nous écrirons, par exemple, la classe **Rectangle** suivante :

```
class Rectangle {

private:

double largeur = 0;
double hauteur = 0;

public:

Rectangle() = default;
Rectangle(double l, double h);
~Rectangle() = default;

double & largeur();
const double & largeur() const;

double & hauteur();
const double & hauteur() const;

double surface() const;
};
```

▸ **Exercice 1.**
Dans le fichier ***Rectangle.cpp****, écrivez l'implémentation des méthodes précédentes et dans un fichier* ***main.cpp****, écrivez la fonction main qui teste votre classe* ***Rectangle*** *avec :*

```
Rectangle r1(2.6, 4.42), r2;
std::cout << r1.surface() << std::endl;
r1.setLargeur(3.9);
```

▸ **Exercice 2.**
On souhaite maintenant représenter des carrés. Écrivez la classe **Carre** *en se basant sur le modèle de la classe précédente. Vous aurez une variable membre cote et les méthodes getCote et setCote. Dans la méthode main, ajoutez et testez :*

```
Carre c1(3.8), c2;
std::cout << c1.surface() << std::endl;
c2.setCote(2.9);
```

En comparant la classe **Carre** à celle de **Rectangle**, on remarque qu'elle lui ressemble énormément. On peut alors se demander s'il est possible de réutiliser la classe **Rectangle** pour définir la classe **Carre**.

La réponse est oui grâce à la notion d'héritage qui permet de définir une classe B, appelée classe dérivée, par spécialisation/extension d'une autre classe A, appelée superclasse (ou classe mère, ou encore classe de base).

Pourquoi est-il naturel de faire hériter la classe **Carre** de **Rectangle** ? Tout simplement parce qu'un carré est-un rectangle particulier (la largeur est égale à la longueur).

Une classe dérivée est donc une spécialisation de sa superclasse. Toutefois, elle peut aussi être vue comme une extension dans le mesure où on pourra ajouter de nouveaux attributs et méthodes dans la classe dérivée. D'une façon générale, à chaque fois que la relation est-un peut être appliquée entre deux classes, la relation d'héritage devra être utilisée. La classe dérivée B est héritière au sens qu'elle possède tous les attributs et les méthodes de sa superclasse A en plus de ses propres attributs et méthodes. Toutefois, elle n'hérite pas des constructeurs, du destructeur, du constructeur de copie, ni de l'opérateur d'affectation. L'en-tête de la déclaration d'une classe B qui dérive d'une classe A, sera :

class B : mode dérivation A

où mode dérivation est soit **private** (comportement par défaut), **protected** ou **public**.

L'héritage est contrôlé par le mode dérivation.

- S'il est **private**, tous les attributs et les méthodes **public** ou **protected** de la superclasse deviennent privés dans la classe dérivée.
- Si le mode est **proctected**, ils deviennent protégés.
- Enfin, si le mode est **public**, tous les attributs et les méthodes de la superclasse gardent leur mode d'accès dans la classe dérivée. Ce dernier mode est le mode de dérivation habituel.

La classe dérivée est donc amenée, si nécessaire, à définir ses propres constructeurs, destructeur, constructeur de copie, et opérateur de copie. Lorsqu'elle définit son constructeur, par défaut le constructeur par défaut de la superclasse est d'abord exécuté. Mais, le constructeur de la classe dérivée peut également faire appel à un constructeur particulier de la superclasse avec la notation : **B(...) : A(...) {}**.
Le constructeur de la classe A sera exécuté **avant** celui de B.

Questions

- Quelle est la syntaxe qui permet de mettre en place l'héritage entre deux classes?
- Quelle est la signification du mot clé "**protected**"?
- Comment est appelé le constructeur de la classe mère?

▸ **Exercice 3.**

- *Récrivez la classe **Carre** en la faisant hériter de la classe **Rectangle** selon le mode **public**. Dans la classe **Rectangle**, vous déclarerez les variables largeur et hauteur **protected**, et vous définirez le constructeur Carre(double c), dans la classe **Carre**, qui initialise le coté du carré.*
- *Compilez et exécutez à nouveau votre fonction main. Constatez que dans l'énoncé c1.surface(), la méthode appliquée à un objet de type **Carre** est obtenue par héritage de la classe **Rectangle**.*

▸ **Exercice 4.**

*Lorsque un objet de la classe dérivée est détruit, le destructeur de cette classe dérivée s'applique d'abord, puis celui de la superclasse. Mettez en évidence ce comportement, en définissant un destructeur dans **Rectangle** et un autre dans **Carre**.*

▸ **Exercice 5.**

*Dans le fichier **Rectangle.cpp**, donnez l'implémentation de la méthode : void quiSuisJe() const; pour qu'elle renvoie la chaîne de caractères "Je suis un rectangle." et testez pour un objet de type Rectangle.*

▸ **Exercice 6. *Redéfiinition de méthodes***

- *Dans votre fonction main, appliquez la méthode quiSuisJe sur r1 et puis sur c1. Compilez et testez votre méthode main. Que constatez-vous ?*

  Une classe dérivée hérite d'attributs et méthodes de sa superclasse, elle peut aussi définir ses propres variables et méthodes membres, mais elle peut également redéfinir

des méthodes héritées. Elle le fait lorsqu'elle désire modifier l'implémentation d'une méthode d'une classe parent, en particulier pour adapter son action à des besoins spécifiques. C'est le cas ici pour la méthode quiSuisJe à redéfinir dans ***Carre****.*

- *Dans la classe* ***Carre****, redéfinissez la méthode quiSuisJe pour qu'elle renvoie la chaîne de caractères "Je suis un carré.".*
- *Recompilez et exécutez votre programme pour obtenir le résultat attendu.*

Note : une méthode peut être spécifiée redéfinie à l'aide du mot-clé **override**.

▸ **Exercice 7.** ***Autres figures géométriques...***

- *Sur le modèle des classes* ***Rectangle*** *et* ***Carre****. Écrivez la classe* ***Ellipse*** *et sa classe dérivée* ***Cercle****. On rappelle que pour une ellipse de petit rayon a, et de grand rayon b, la surface est πab.*
- *Les quatre classes* ***Rectangle****,* ***Carre****,* ***Ellipse*** *et* ***Cercle*** *sont des figures géométriques, et il peut être judicieux de définir une classe de base* ***Figure*** *pour les représenter.*
- *Dessinez le graphe d'héritage de ces 5 classes.*
- *Dans la fonction main déclarez une variable f de type* ***Figure*** *et appliquez la méthode quiSuisJe.*

▸ **Exercice 8.**

- *Placer dans la fonction main les deux déclarations suivantes, puis compiler :*

```
Rectangle  r1;
Rectangle  r2 = Carre (8);
```

Est ce qu'une variable de type Rectangle peut désigner un objet de type Carre?

- *Remplacer la déclaration de la variable r2 par :*

```
Rectangle  r2 = Cercle (8);
```

Est ce qu'une variable de type Rectangle peut désigner un objet de type Cercle?

- *Remplacer la déclaration de la variable r2 par :*

```
Carre  r2 = r1;
```

  - *Est ce qu'on peut affecter à un objet de type plus spécifique (sousclasse) un autre objet de type plus général (superclasse) ?*

▸ **Exercice 9.** ***Liaison dynamique et méthodes virtuelles***

Dans votre fichier main.ccp, ajoutez la fonction suivante :

```
void afficher(Figure *f){
f->quiSuisJe();
}
```

puis dans votre fonction main, testez le code suivant :

```
Rectangle *r = new Rectangle(1,8);
afficher(r);

Ellipse *e = new Ellipse(11,1);
afficher(e);
```

- *Commenter le résultat obtenu (En toute bonne logique, que devriez-vous obtenir) ?*

Le problème vient du fait que la méthode à appliquer est déterminée de façon statique (on parle de Liaison statique), c'est-à- dire à la compilation. Ainsi, puisque dans la fonction afficher, le paramètre est de type Figure, la méthode quiSuisJe choisie par le compilateur est celle de la classe Figure et non pas celle du paramètre effectif qui n'est connu que de façon dynamique, c'est-à-dire à l'exécution du programme.*

La solution ? Le langage C++ met en place le mécanisme de liaison dynamique de telle sorte que la méthode appliquée soit celle de l'objet qui est effectivement utilisée au moment de son exécution.

En C++, les méthodes sur lesquelles la liaison dynamique pourra s'appliquer devront être déclarées ***virtual****. Pour cela, on met le mot-clé* ***virtual*** *devant la toute première définition de la méthode. Toutes les méthodes redéfinies dans les classes dérivées deviennent alors virtuelles.*

- *Dans la classe Figure, rendez la méthode quiSuisJe virtuelle, recompilez votre programme et testez-le pour obtenir le comportement attendu.*

D'autre part, quand le mécanisme de liaison dynamique est mis en jeu, pour que le destructeur de l'objet effectivement traité soit appliquée, il est impératif qu'il soit également déclaré virtual dans la superclasse initiale.

▸ **Exercice 10.**

- *Dans la fonction main, déclarez une variable listFig de type vecteur (classe vector) de Figure *, et ajoutez dans ce vecteur plusieurs rectangles, carrés, ellipses et cercles.*
- *Parcourez le vecteur pour afficher sur la sortie standard la nature de chaque figure et sa surface. Quel est le problème ?*

Il faut ajouter la méthode virtuelle surface dans la classe Figure. Mais quelles instructions mettre dans le corps de cette méthode, puisque on n'est pas capable de déterminer la surface du figure quelconque ?

La solution est ne pas l'écrire. Pour cela, on définit la méthode de façon virtuelle pure, en ajoutant =0 à la fin du prototype de la méthode.

- *Rendez surface* ***virtuelle pure*** *et testez votre programme.*