

MOOC Intro POO C++

Tutoriels semaine 4 : Héritage

Les tutoriels sont des exercices qui reprennent des exemples similaires à ceux du cours et dont le corrigé est donné progressivement au fur et à mesure de l'exercice lui-même.

Ils sont conseillés comme un premier exercice sur un sujet que l'étudiant ne pense pas encore assez maîtriser pour aborder par lui-même un exercice « classique ».

Les solutions sont fournies au fur et à mesure sur les pages paires.

Cet exercice correspond à l'exercice « pas à pas » page 134 de l'ouvrage *C++ par la pratique (3e édition, PPUR)*.

Introduction

Le but de cet exercice est d'illustrer la notion d'héritage en utilisant une hiérarchie de figures géométriques.

Dans le fichier `figures.cc`, commencez par définir des classes `Rectangle` et `Cercle`. Munissez-les d'attributs adaptés et d'une méthode `surface`.

Essayez de le faire par vous-même sans regarder la solution (page suivante).

Solution :

```
#include <cmath>
using namespace std;

// -----
class Rectangle {
private:
    double largeur;
    double longueur;
public:
    double surface() const { return largeur * longueur; }
    double getLongueur() const { return longueur; }
    double getLargeur() const { return largeur; }
    void setLargeur(double l) { largeur = l; }
    void setLongueur(double l) { longueur = l; }
};

// -----
class Cercle {
public:
    double surface() const { return M_PI * rayon * rayon; }
    double getRayon() const { return rayon; }
    void setRayon(double r) {
        if (r < 0.0) r = 0.0;
        rayon = r;
    }
private:
    double rayon;
};
```

Ajoutons ensuite une classe « rectangle coloré ».

Cette classe doit uniquement avoir un attribut de plus, couleur, disons de type unsigned int.

Essayez de le faire par vous-même sans regarder la solution (page suivante).

Solution : Ceci se fait très simplement :

1. La classe RectangleColore hérite de Rectangle :

```
class RectangleColore : public Rectangle {  
};
```

2. et a/possède un attribut de plus :

```
class RectangleColore : public Rectangle {  
protected:  
    unsigned int couleur;  
};
```

3. On peut également vouloir changer le statut des attributs hérités de Rectangle en les passant de private à protected :

```
class Rectangle {  
protected:  
    double largeur;  
    double longueur;  
public:  
    // ... comme avant
```

Pour rendre ces classes un peu plus réalistes (et utilisables) ajoutons-leur un constructeur à chacune (si ce n'est pas déjà fait) :

```
class Rectangle {
protected:
    double largeur;
    double longueur;
public:
    Rectangle(double larg, double L) : largeur(larg), longueur(L) {}
    // ... comme avant
    ...

class RectangleColore : public Rectangle {
protected:
    unsigned int couleur;
public:
    RectangleColore(double larg, double L, unsigned int c)
        : Rectangle(larg, L), couleur(c) {}
};
```

Testez avec ce main simpliste :

```
int main() {
    RectangleColore r(4.3, 12.5, 4);
    cout << r.getLargeur() << endl;
    return 0;
}
```

Continuez en suivant les points ci-dessous :

- La classe Figure, contenant deux attributs x et y (représentant le centre de la figure), une méthode affiche(ostream&) qui affiche simplement les coordonnées du centre et un constructeur prenant les coordonnées du centre.
- Faites hériter les classes Cercle et Rectangle de la classe Figure (il faut donc modifier en particulier la classe Cercle).

Essayez de le faire par vous-même sans regarder la solution (page suivante).

Solution : Pour la classe Figure, rien de particulier, et il suffit de déplacer les définitions correspondantes de la classe Cercle à la classe Figure (donc les supprimer de Cercle) :

```
class Figure {
public:
    void getPoint(double& abscisse, double& ordonnee) const {
        abscisse = x;
        ordonnee = y;
    }
    void setPoint(double abscisse, double ordonnee) {
        x = abscisse;
        y = ordonnee;
    }
protected:
    double x; // abscisse du point de reference
    double y; // ordonnee du point de reference
};
```

puis on ajoute la méthode affiche et le constructeur (rien de spécial ici) :

```
class Figure {
public:
    Figure(double x = 0.0, double y = 0.0) : x(x), y(y) {}
    void affiche(ostream& sortie) {
        sortie << "centre = (" << x << ", " << y << ")";
    }
    //... suite comme avant
};
```

Venons-en à l'héritage proprement dit. Il suffit simplement d'ajouter la « marque » de l'héritage aux deux classes concernées.

```
class Rectangle : public Figure { ...
...
class Cercle : public Figure { ...
```

On peut maintenant tester que l'on hérite bien des propriétés de la classe Figure :

```
int main() {  
    RectangleColore r(4.3, 12.5, 4);  
    cout << r.getLargeur() << endl;  
    r.affiche(cout);  
    cout << endl;  
  
    Cercle c;  
    c.setPoint(2.3, 4.5);  
    c.setRayon(12.2);  
    c.affiche(cout);  
    cout << endl;  
    return 0;  
}
```

Si vous êtes perdus, voici ici le code complet à ce stade.

[Testez ce programme : compilez-le, exécutez-le. Essayez ensuite d'autres utilisations pour bien comprendre.]

Évidemment pour être plus intéressantes, il faudrait répercuter les possibilités de la classe Figure au niveau des constructeurs des classes Rectangle et Cercle, et éventuellement enrichir la méthode affiche.

(Solution page suivante.)

Solution :

Pour les constructeurs c'est assez facile :

```
class Rectangle : public Figure {
    ...
    Rectangle(double larg, double L, double x, double y)
        : Figure(x,y), largeur(larg), longueur(L) {}
    ...
};

class Cercle : public Figure {
public:
    Cercle(double rayon, double x = 0.0, double y = 0.0)
        : Figure(x,y), rayon(rayon) {}
    ...
};
```

Pour la méthode affiche c'est un peu plus subtil. Supposons que l'on souhaite que, pour la classe Cercle, la méthode affiche également le rayon (mais continue d'afficher le centre).

On veut donc faire quelque chose comme (syntaxe fantaisiste) :

« cercle.affiche() = figure.affiche() + cout « rayon »

Cela se fait exactement de cette façon mais en utilisant la syntaxe du C++ et plus exactement l'opérateur de résolution de portée :

```
void Cercle::affiche(ostream& sortie) {
    Figure::affiche(sortie);
    sortie << " , r=" << rayon;
}
```

et ne pas oublier de définir la méthode dans la classe Cercle :

```
class Cercle : public Figure {
public:
    Cercle(double rayon, double x = 0.0, double y = 0.0)
        : Figure(x,y), rayon(rayon) {}
    void affiche(ostream&);
    ...
};
```

On peut bien sûr faire de même avec la classe Rectangle :

```
void Rectangle::affiche(ostream& sortie) {
    Figure::affiche(sortie);
    sortie << " , largeur=" << largeur
    << " , longueur=" << longueur;
}
```

Testez le programme avec le main suivant :

```
int main() {  
    RectangleColore r(4.3, 12.5, 4);  
    cout << r.getLargeur() << endl;  
    r.affiche(cout);  
    cout << endl;  
  
    Cercle c(12.2, 2.3, 4.5);  
    c.affiche(cout);  
    cout << endl;  
  
    Rectangle r2(1.2, 3.4, 12.3, 43.2);  
    r2.affiche(cout);  
    cout << endl;  
  
    return 0;  
}
```
