MOOC Intro POO C++

Corrigés semaine 5

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 16 : Formes polymorphiques

Cet exercice correspond à l'exercice n°60 (pages 150 et 335) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

1.1 Formes :

Définissez une classe `Forme` *en la dotant d'une méthode qui affiche* [...]

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

class Forme {
public:
  void description() const {
    cout << "Ceci est une forme." << endl;
  }
};
```

Ajoutez au programme une classe `Cercle` *héritant de la classe* `Forme`*, et possédant une méthode* `void description()` *qui affiche* [...]

```
class Cercle : public Forme {
public:
  void description() const {
    cout << "Ceci est un cercle." << endl;
  }
};
```

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

La différence vient de ce que `c.description();` appelle la méthode `description()` de la classe `Cercle` (c'est-à-dire `Cercle::description()`), alors que `f2.description();` appelle celle de la classe `Forme` (c'est-à-dire `Forme::description()`), bien qu'elle ait été construite par une copie d'un cercle.

Le polymorphisme n'opère pas ici car **aucune** des deux conditions nécessaires n'est remplie : la méthode n'est pas virtuelle et on ne passe pas par des références ni pointeurs.

[...] *Ajoutez encore au programme une fonction* `void affichageDesc(Forme& f)` [...]
Le résultat vous semble-t-il satisfaisant ?

Avec cette fonction, nous apportons une solution au second aspect puisqu'en effet nous passons l'argument par référence.

Le résultat n'est cependant toujours pas satisfaisant (c'est toujours la méthode `Forme::description()` qui est appelée) car le premier problème subsiste : la méthode n'est pas virtuelle.

Modifiez le programme (ajoutez 1 seul mot) pour que le résultat soit plus conforme à ce que l'on pourrait attendre.

Il suffit donc d'ajouter **`virtual`** devant le prototype de la méthode `description` de la classe `Forme`.

Voici le programme complet :

```cpp
#include <iostream>
using namespace std;

class Forme {
public:
  virtual void description() const {
    cout << "Ceci est une forme." << endl;
  }
};

class Cercle : public Forme {
public:
  void description() const {
    cout << "Ceci est un cercle." << endl;
  }
};

void affichageDesc(const Forme& f) { f.description(); }

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

1.2 Formes abstraites

Modifiez la classe `Forme` *de manière à en faire une classe abstraite* [...]

```cpp
class Forme {
public:
  virtual void description() const {
    cout << "Ceci est une forme !" << endl;
  }
  virtual double aire() const = 0;
};
```

Ce qui en fait une méthode virtuelle pure c'est le `=0` derrière qui indique que pour cette classe cette méthode ne sera pas implémentée (i.e. pas de définition, c.-à-d. pas de corps).

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

```cpp
class Cercle : public Forme {
public:
  Cercle(double r = 0.0) : rayon(r) {}
  void description() const {
    cout << "Ceci est un cercle !" << endl;
  }
  double aire() const { return M_PI * rayon * rayon; }
private:
  double rayon;
};

```

```cpp
class Triangle : public Forme {
public:
  Triangle(double h = 0.0, double b = 0.0) : base(b), hauteur(h) {}
  void description() const {
    cout << "Ceci est un triangle !" << endl;
  }
  double aire() const { return 0.5 * base * hauteur; }
private:
  double base; double hauteur;
};
```

Modifiez la fonction `affichageDesc` *pour qu'elle affiche, en plus, l'aire* [...]

```cpp
void affichageDesc(Forme& f) {
  f.description();
  cout << " son aire est " << f.aire() << endl;
}
```

et le programme complet :

```cpp
#include <iostream>
#include <cmath> // pour M_PI
using namespace std;

class Forme {
public:
  virtual void description() const {
    cout << "Ceci est une forme !" << endl;
  }
  virtual double aire() const = 0;
};

class Cercle : public Forme {
public:
  Cercle(double r = 0.0) : rayon(r) {}
  void description() const {
    cout << "Ceci est un cercle !" << endl;
  }
  double aire() const { return M_PI * rayon * rayon; }
private:
  double rayon;
};

class Triangle : public Forme {
public:
  Triangle(double h = 0.0, double b = 0.0) : base(b), hauteur(h) {}
  void description() const {
    cout << "Ceci est un triangle !" << endl;
  }
  double aire() const { return 0.5 * base * hauteur; }
private:
  double base; double hauteur;
};

void affichageDesc(Forme& f) {
  f.description();
  cout << " son aire est " << f.aire() << endl;
}

int main()
{
  Cercle c(5);
  Triangle t(10, 2);
  affichageDesc(t);
```

```
45    affichageDesc(c);
46    return 0;
47  }
```

qui donne comme résultat :

```
1  Ceci est un triangle !
2   son aire est 10
3  Ceci est un cercle !
4   son aire est 78.5398
```

Exercice 17 : Encore plus de formes

Cet exercice correspond à l'exercice n°60 (pages 149 et 334) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Prototypez et définissez les classes [...] `Figure`

```cpp
#include <iostream>
using namespace std;

class Figure {
public:
  virtual void affiche () const = 0;
  virtual Figure* copie() const = 0;
};
```

[...] *Trois sous-classes (héritage publique) de* `Figure` : `Cercle`, `Carre` *et* `Triangle`.

```cpp
class Cercle : public Figure {
};

class Carre : public Figure {
};

class Triangle : public Figure {
};
```

Une classe nommée `Dessin`, *qui modélise une collection de figures.* [...]

```cpp
class Dessin {
public:
  void ajouteFigure();
private:
  vector<Figure*> contenu;
};
```

Comme conseillé en cours, plutôt que d'hériter de la classe `vector`, on préfère encapsuler la collection dans la classe.

Pour les pointeurs, nous avons ici choisi des pointeurs «à la C». En C++11, on pourrait plutôt opter pour des `unique_prt`. Une solution dans ce sens est proposée à la fin.

[...] *définissez les attributs requis pour modéliser les objets correspondant*

```cpp
class Cercle : public Figure {
private:
  double rayon;
};

class Carre : public Figure {
private:
  double cote;
};

class Triangle : public Figure {
private:
  double base; double hauteur;
};
```

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

```cpp
class Cercle : public Figure {
public:
```

```
  Cercle(double r = 0.0) : rayon(r) {
    cout << "Construction d'un cercle de rayon " << rayon << endl;
  }

  Cercle(const Cercle& autre) : Figure(autre), rayon(autre.rayon) {
    cout << "Copie d'un cercle de rayon " << rayon << endl;
  }

  ~Cercle() { cout << "Destruction d'un cercle" << endl; }

private:
  double rayon;
};

//---------------------------------------------------------------------
class Carre : public Figure {
public:
  Carre(double x = 0.0) : cote(x) {
    cout << "Construction d'un carre de cote " << cote << endl;
  }

  Carre(const Carre& autre) : Figure(autre), cote(autre.cote) {
    cout << "Copie d'un carre de cote " << cote << endl;
  }

  ~Carre() { cout << "Destruction d'un carre" << endl; }

private:
  double cote;
};

//---------------------------------------------------------------------
class Triangle : public Figure {
public:
  Triangle(double b = 0.0, double h = 0.0) : base(b), hauteur(h) {
    cout << "Construction d'un triangle " << base << "x" << hauteur << endl;
  }

  Triangle(const Triangle& autre) : Figure(autre)
                                  , base(autre.base), hauteur(autre.hauteur) {
    cout << "Copie d'un triangle " << base << "x" << hauteur << endl;
  }

  ~Triangle() { cout << "Destruction d'un triangle" << endl; }

private:
  double base; double hauteur;
};
```

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

Cette partie, quoique finalement simple, est peut être d'un abord plus difficile. Ne vous laissez pas démonter par l'apparente difficulté conceptuelle et, une fois de plus, décomposez le problème :

- Que veut-on ?

  Retourner le pointeur sur une copie de l'objet :

  ```
  Figure* copie() const { }
  ```

  (jusque là c'était déjà donné dans l'énoncé au niveau de `Figure`).
- comment fait-on une copie ?

  En utilisant le constructeur de copie qu'il dit le Monsieur...

Par exemple pour la classe `Cercle`, cela s'écrit `Cercle(...)`

— De qui fait-on une copie ?

de nous-même.

Comment ça s'écrit "nous-même" ?

`*this` (contenu de l'objet pointé par `this`).

On a donc : `Cercle(*this)`

— Que veut-on de plus ?

Que la copie soit effectivement placée en mémoire et on veut en retourner l'adresse (allocation dynamique).

Cela se fait avec `new`

Et donc finalement on aboutit à :

```
class Cercle : public Figure {
...
  Figure* copie() const { return new Cercle(*this); }
...
};

class Carre : public Figure {
...
  Figure* copie() const { return new Carre(*this); }
...
};

class Triangle : public Figure {
...
  Figure* copie() const { return new Triangle(*this); }
...
};
```

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

Trivial :

```
class Cercle : public Figure {
...
void affiche() const {
    cout << "Un cercle de rayon " << rayon << endl;
  }
...
};

class Carre : public Figure {
...
  void affiche() const {
    cout << "Un carre de de cote " << cote << endl;
  }
...
};

class Triangle : public Figure {
...
  void affiche() const {
    cout << "Un triangle " << base << "x" << hauteur << endl;
  }
...
};
```

Ajoutez un destructeur explicite pour la classe `Dessin` [...]

```
~Dessin() {
```

```
  cout << "Le dessins s'efface..." << endl;
  for (auto & fig : contenu) delete fig;
}
```

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

```
public:
  ~Dessin() {
    cout << "Le dessins s'efface..." << endl;
    for (unsigned int i(0); i < contenu.size(); ++i) delete contenu[i];
  }
  void ajouteFigure(const Figure& fig) {
    contenu.push_back(fig.copie());
  }
  void affiche() const {
    cout << "Je contiens :" << endl;
    for (unsigned int i(0); i < contenu.size(); ++i) {
      contenu[i]->affiche();
    }
  }
private:
  vector<Figure*> contenu;
};
```

Votre programme devrait indiquer que le destructeur du dessin est invoqué... mais pas les destructeurs des figures stockées dans le dessin. Pourquoi ?

Car le destructeur n'est pas virtuel. Le compilateur vous a d'ailleurs averti par les warnings.

Pour y remédier, il suffit d'ajouter le mot clef **`virtual`** devant.

Voici le code complet :

```
#include <iostream>
#include <vector>
using namespace std;

//---------------------------------------------------------------------
class Figure {
public:
  virtual void affiche() const = 0;
  virtual Figure* copie() const = 0;
  virtual ~Figure() { cout << "Une figure de moins." << endl; }
};

//---------------------------------------------------------------------
class Cercle : public Figure {
public:
  Cercle(double x = 0.0) : rayon(x) {
    cout << "Construction d'un cercle de rayon " << rayon << endl;
  }

  Cercle(const Cercle& autre) : Figure(autre), rayon(autre.rayon) {
    cout << "Copie d'un cercle de rayon " << rayon << endl;
  }

  ~Cercle() { cout << "Destruction d'un cercle" << endl; }

  Figure* copie() const { return new Cercle(*this); }

  void affiche() const {
    cout << "Un cercle de rayon " << rayon << endl;
  }

private:
```

```
  double rayon;
};

//---------------------------------------------------------------------
class Carre : public Figure {
public:
  Carre(double a = 0.0) : cote(a) {
    cout << "Construction d'un carre de cote " << cote << endl;
  }

  Carre(const Carre& autre) : Figure(autre), cote(autre.cote) {
    cout << "Copie d'un carre de cote " << cote << endl;
  }

  ~Carre() { cout << "Destruction d'un carre" << endl; }

  Figure* copie() const { return new Carre(*this); }

  void affiche() const {
    cout << "Un carre de cote " << cote << endl;
  }

private:
  double cote;
};

//---------------------------------------------------------------------
class Triangle : public Figure {
public:
  Triangle(double b = 0.0, double h = 0.0) : base(b), hauteur(h) {
    cout << "Construction d'un triangle " << base << "x" << hauteur << endl;
  }

  Triangle(const Triangle& autre)
    : Figure(autre), base(autre.base), hauteur(autre.hauteur)
  {
    cout << "Copie d'un triangle " << base << "x" << hauteur << endl;
  }

  ~Triangle() { cout << "Destruction d'un triangle" << endl; }

  Figure* copie() const { return new Triangle(*this); }

  void affiche() const {
    cout << "Un triangle " << base << "x" << hauteur << endl;
  }

private:
  double base; double hauteur;
};

//---------------------------------------------------------------------
class Dessin {
public:
  ~Dessin() {
    cout << "Le dessins s'efface..." << endl;
    for (unsigned int i(0); i < contenu.size(); ++i) delete contenu[i];
  }
  void ajouteFigure(const Figure& fig) {
    contenu.push_back(fig.copie());
  }
  void affiche() const {
    cout << "Je contiens :" << endl;
```

```
    for (unsigned int i(0); i < contenu.size(); ++i) {
      contenu[i]->affiche();
    }
  }
private:
  vector<Figure*> contenu;
};

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

int main()
{
  Dessin dessin;

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

  unCercleDePlus(dessin); // impossible

  cout << endl << "Affichage du dessin : " << endl;
  dessin.affiche();
  return 0;
}
```

[...] *le système doit interrompre prématurément votre programme, en vous adressant un message hargneux* [...] *Quel est, à votre avis, le motif pour un tel comportement ?*

Ceci est le cas classique de la libération incongrue de mémoire par une copie passagère de l'objet. Voir le cours pour plus de détails, mais en deux mot, `tmp` crée une copie de `img` qui est détruite à la fin de la fonction et libère la mémoire pointée, laquelle est encore utilisée par l'objet passé en argument comme `img`. La prochaine tentative d'accès à cette mémoire via cet objet (du `main()`) provoque une erreur comme indiquée.

Pour y palier, il faudrait définir le constructeur de copie de `Dessin` afin de faire une copie *profonde*.

Voici enfin une version en C++11 avec `unique_prt` (pour varier).

Notez comme l'emploi des `unique_prt` résoud de façon drastique le problème précédent : on ne peut simplement plus faire de copie de `Dessin`.

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

class Figure {
public:
  virtual void affiche() const = 0;
  virtual unique_ptr<Figure> copie() const = 0;
  virtual ~Figure() { cout << "Une figure de moins." << endl; }
};

class Cercle : public Figure {
public:
  Cercle(double r = 0.0) : rayon(r) {
    cout << "Et hop, un cercle de plus !" << endl;
  }
  Cercle(const Cercle& autre) : rayon(autre.rayon) {
    cout << "Et encore un cercle qui fait des petits !" << endl;
```

```
  }
  ~Cercle() { cout << "le dernier cercle ?" << endl; }
  unique_ptr<Figure> copie() const override {
    return unique_ptr<Figure>(new Cercle(*this)); }
  void affiche() const override {
    cout << "Un cercle de rayon " << rayon << endl;
  }
private:
  double rayon;
};

class Carre : public Figure {
public:
  Carre(double a = 0.0) : cote(a) {
    cout << "Coucou, un carre de plus !" << endl;
  }
  Carre(const Carre& autre) : cote(autre.cote) {
    cout << "Et encore un carre qui fait des petits !" << endl;
  }
  ~Carre() { cout << "bou... un carre de moins." << endl; }
  unique_ptr<Figure> copie() const override {
    return unique_ptr<Figure>(new Carre(*this)); }
  void affiche() const override {
    cout << "Un carre de de cote " << cote << endl;
  }
private:
  double cote;
};

class Triangle : public Figure {
public:
  Triangle(double h = 0.0, double b = 0.0)
    : base(b), hauteur(h)
  {
    cout << "Un Triangle est arrive !" << endl;
  }
  Triangle(const Triangle& autre) : base(autre.base), hauteur(autre.hauteur)
  {
    cout << "Et encore un triangle qui fait des petits !" << endl;
  }
  ~Triangle() { cout << "La fin du triangle." << endl; }
  unique_ptr<Figure> copie() const override {
    return unique_ptr<Figure>(new Triangle(*this)); }
  void affiche() const override {
    cout << "Un triangle " << base << "x" << hauteur << endl;
  }
private:
  double base; double hauteur;
};

class Dessin {
public:
  ~Dessin() { cout << "Le dessins s'efface..." << endl; }
  void ajouteFigure(const Figure& fig) { contenu.push_back(fig.copie()); }
  void affiche() const {
    cout << "Je contiens :" << endl;
    for (auto const& fig : contenu) { fig->affiche(); }
  }
private:
  vector<unique_ptr<Figure>> contenu;
};

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();
  return 0;
}
```

Exercice 18 : Librairie

Cet exercice correspond à l'exercice n°72 (pages 183 et 381) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

/* La classe Livre */

class Livre {
public:
  Livre(string, string, int, bool);
  virtual ~Livre() {}
  virtual double calculer_prix() const ;
  virtual void afficher() const ;

protected:
  string titre ;
  string auteur ;
  int nbPages ;
  bool bestseller ;
};

// constructeur

Livre::Livre(string titre, string auteur,
             int nbPages, bool bestseller)
  : titre(titre), auteur(auteur), nbPages(nbPages),
    bestseller(bestseller)
{}

// calcul du prix d'un livre
double Livre::calculer_prix() const
{
  double p(nbPages* 0.3);
  if (bestseller)
    return (p + 50.0);
  else return p;
}

// affichage des caracteristiques d'un livre
void Livre::afficher() const
{
  cout << "titre : " << titre <<endl;
  cout << "auteur : " << auteur <<endl;
  cout << "nombre de pages : " << nbPages <<endl;
  cout << "bestseller : " ;
  if (bestseller) cout << "oui";
  else cout << "non";
  cout << endl;
  cout << "prix : " << calculer_prix() << endl;
}

/* la sous-classe Roman*/

class Roman: public Livre {
public:
  Roman(string, string, int, bool, bool);
```

```
  virtual ~Roman() {}
  virtual void afficher() const ;

protected:
  bool biographie;

};

Roman::Roman(string titre, string auteur,
             int nbPages, bool bestseller, bool biographie)
  : Livre(titre, auteur, nbPages, bestseller), biographie(biographie)

{}

void Roman::afficher() const
{
  Livre::afficher();
  if (biographie)
    cout << "Ce roman est une biographie" << endl;
  else
    cout << "Ce roman n'est pas une biographie" << endl;
}

/* les romans policiers */

class Policier : public Roman {
public:
  using Roman::Roman; // heritage du constructeur (on n'a pas de nouvel attribut)
  Policier(string, string, int, bool, bool);
  virtual ~Policier() {}
  virtual double calculer_prix() const ;

};

double Policier::calculer_prix() const
{
  double p (Livre::calculer_prix() - 10.0);
  if (p <= 0.0) p = 1.0;
  return p;
}

/* la sous-classe BeauLivre */

class BeauLivre : public Livre {
public:
  using Livre::Livre; // heritage du constructeur (on n'a pas de nouvel attribut)
  virtual ~BeauLivre() {}
  virtual double calculer_prix() const ;
};

double BeauLivre::calculer_prix() const
{
  return (Livre::calculer_prix() + 30.0);
}

/* la classe Librairie */

class Librairie {
public:
  void ajouter_livre(Livre* livre) { contenu.push_back(livre); }
  void vider_stock();
  void afficher() const;
private:
```

```
  vector <Livre*> contenu;
};

void Librairie::afficher() const
{
  for (auto livre : contenu) {
    livre->afficher();
    cout << endl;
  }
}

void Librairie::vider_stock()
{
  for (auto livre : contenu) { delete livre; }
  contenu.clear();
}

int main()
{
  Librairie l;

  l.ajouter_livre(new Policier("Le chien des Baskerville", "A.C.Doyle",
                               221, false, false));
  l.ajouter_livre(new Policier("Le Parrain ", "A.Cuso",
                               367, true, false));
  l.ajouter_livre(new Roman("Le baron perche", "I. Calvino",
                            283, false, false));
  l.ajouter_livre(new Roman ("Memoires de Geronimo", "S.M. Barrett",
                             173, false, true));
  l.ajouter_livre(new BeauLivre ("Fleuves d'Europe", "C. Osborne",
                                 150, false));

  l.afficher();
  l.vider_stock();

  return 0;
}
```