Choisissez votre style : **colorisé**, **impression**

Correction 7
Polymorphisme

Exercice 1 : Formes polymorphiques (niveau 1)

1.1 Formes :

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

```
#include <iostream> // pour cout
using namespace std;   // pour ne pas écrire std::cout et std::endl

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 vu 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 que 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 par 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 :

```
1  #include <iostream>
2  using namespace std;
3
4  class Forme {
5  public:
6    virtual void description() const {
7      cout << "Ceci est une forme !" << endl;
8    }
9  };
10
11 class Cercle : public Forme {
12 public:
13   void description() const {
14     cout << "Ceci est un cercle !" << endl;
15   }
16 };
17
18 void affichageDesc(Forme& f) { f.description(); }
19
20 int main()
21 {
22   Forme f;
23   Cercle c;
24   // f.description();
25   // c.description();
26   // Forme f2(c);
27   // f2.description();
28   affichageDesc(f);
29   affichageDesc(c);
30   return 0;
31 }
```

1.2 Formes abstraites

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

```
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()`. [...]

```
class Cercle : public Forme {
public:
  Cercle(double x = 0.0) { rayon = x; }
  void description() const {
    cout << "Ceci est un cercle !" << endl;
  }
  double aire() const { return 3.141592653 * 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;
};
```

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

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

et le programme complet :

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

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

class Cercle : public Forme {
public:
  Cercle(double x = 0.0) { rayon = x; }
  void description() const {
    cout << "Ceci est un cercle !" << endl;
  }
  double aire() const { return 3.141592653 * 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);
  affichageDesc(c);
  return 0;
}
```

qui donne comme résultat :

```
Ceci est un triangle !
 son aire est 10
Ceci est un cercle !
 son aire est 78.5398
```

Exercice 2 : encore des vaccins (niveau 1, polymorphisme)

Voici une reprise du code de la série 8 à laquelle a été ajoutée la classe `Compagnie` :

```
#include <string>
#include <vector>
#include<iostream>

using namespace std;

// prix du conditionnement d'une unité
const double COND_UNITE(0.5);

// prix de base de fabrication d'une unité
const double PRIX_BASE(1.5);

// majoration du prix de fabrication pour vaccin "high tech"
const double MAJORATION_HIGHTECH(0.5);

// reduction du cout du à la delocalisation
const double REDUCTION_DELOC(0.2);

enum Fabrication {Standard, HighTech};

/************************************
 * Un classe pour représenter un vaccin
 ************************************/

class Vaccin{
public:

  Vaccin(string _nom, double _volume_dose, unsigned int _nb_doses,
         Fabrication _fabrication = Standard)
    :nom(_nom), volume_dose(_volume_dose), nb_doses(_nb_doses),
     mode_fabrication(_fabrication)
    {}

  virtual ~Vaccin()
  {}
    // surcharge de l'opérateur << pour afficher les
    // données relatives à un vaccin
    friend ostream& operator<< (ostream& out, const Vaccin& v)
        {
            out << v.nom << endl;
            out << "volume/dose : " << v.volume_dose << endl;
            out << "nombre de doses: " << v.nb_doses << endl;
            out << "mode de fabrication ";
            if (v.mode_fabrication == HighTech)
                out << "haute technologie" << endl;
            else out << "standard" << endl;
            return out;
        }

    // méthode du calcul du cout de conditionnement
    virtual double conditionnement() const{
      return (volume_dose * nb_doses) * COND_UNITE;
  }

    // méthode du calcul du cout de fabrication

    virtual double fabrication() const
        {
            double prix(volume_dose * nb_doses * PRIX_BASE);
            if (mode_fabrication == HighTech)
            {
                prix += prix * MAJORATION_HIGHTECH;

            }
            return prix;
        }

    // méthode du calcul du cout de production

    virtual double production() const
        {

            return fabrication()+conditionnement();
        }

private:
    // nom du vaccin
    string nom;
    // volume par dose de vaccin
    double volume_dose;
    // nombre de doses
    unsigned int nb_doses;
    // mode de fabrication du vaccin
    Fabrication mode_fabrication;


};

/****************************************
 * Une classe pour représenter un vaccin
 * pouvant etre produit de façon délocalisée
 ****************************************/
class Delocalise: public Vaccin{
public:
  Delocalise(string _nom, double _volume_dose,
          unsigned int _nb_doses, Fabrication _fabrication,
          bool  _frontalier)
    :Vaccin(_nom,_volume_dose , _nb_doses, _fabrication),
     frontalier(_frontalier)  {}

  virtual ~Delocalise()
  {}

    // masquage de la méthode héritée de Vaccin
```

```
105     double production() const
106         {
107             double prix = Vaccin::production();
108             if (frontalier)
109             {
110                 prix -= prix * REDUCTION_DELOC;
111             }
112             else
113             {
114                 prix /= 2;
115             }
116             return prix;
117         }
118
119 private:
120     // indique si la production est délocalisée
121     // dans un pays frontalier ou non
122     bool frontalier;
123 };
124
125 // un nouveau type pour représenter un vecteur de Vaccin*
126
127 typedef vector<Vaccin*> Stock;
128
129
130 /************************************************
131  * Une classe pour modéliser une entreprise
132  * pharmaceutique
133  ************************************************/
134 // on aurait aussi pu faire hériter Compagnie de vector<Vaccin*>
135 class Compagnie {
136 public:
137     Compagnie(string _nom)
138         :nom(_nom)
139         {}
140
141     // ajoute un vaccin produit au stock
142     void produire(Vaccin* v) {vaccins.push_back(v); }
143
144     // calcule le cout de production de l'ensemble des vaccins
145     double calculer_cout() const;
146
147     //vide le stock de vaccins
148     void vider_stock() { vaccins.clear();
149     }
150     // détruit tous les vaccins
151     void supprimer_vaccins();
152
153     // affiche tous les vaccins
154     void afficher() const
155         {
156             for (int i(0); i < vaccins.size(); ++i)
157             {
158                 cout << *vaccins[i] << endl;
159
160             }
161         }
162
163
164
165 private:
166     Stock vaccins;
167     string nom;
168 };
169
170 void Compagnie::supprimer_vaccins(){
171   for(unsigned int v(0); v < vaccins.size(); ++v)
172       delete vaccins[v];
173   vider_stock();
174 }
175 double Compagnie::calculer_cout() const
176 {
177     double prix(0);
178     for(unsigned int v(0); v < vaccins.size(); ++v){
179         prix += vaccins[v]->production();
180     }
181
182     return prix;
183 }
184
185
186 // ========================================================================
187 // un petit main pour tester tout ca
188 int main() {
189
190     //Test des parties de la serie 8
191
192     cout << "test de la partie 4.1 " << endl;
193
194
195     Vaccin v1("Zamiflu", 0.55, 200000, HighTech);
196     Vaccin v2("Triphas", 0.20 , 10000);
197     // affichage des vaccins
198     cout << v1 << endl;
199     cout << v2 << endl;
200
201     cout << "le  cout deproduction de  v1 et v2 est : ";
202     cout << v1.production() + v2.production() << endl;
203
204     cout << "test des parties suivantes ..." << endl;
205
206     Delocalise v3("Zamiflu", 0.55, 15000, HighTech, false);
207     Delocalise v4("Triphas", 0.20, 15000, Standard, true);
208     cout << "le  cout de production de  v3 et v4 est : ";
209     cout << v3.production() + v4.production() << endl;
210
211     // test de la partie de cette semaine
212
213     Compagnie c("ICIBA");
214
```

```
    c.produire(&v1);
    c.produire(&v2);
        c.produire(&v3);
    c.produire(&v4);
    cout << endl;
    cout << "Coûts de production de l'ensemble du stock:" << endl;
    c.afficher();
    cout <<"le  cout de production a été de : " <<  c.calculer_cout() << endl;


  return 0;
}
```

Exercice 3 : encore des figures géométriques (polymorphisme, niveau 2)

Prototypez et définissez les classes [...]

```
#include <iostream> // pour cout
using namespace std;   // pour écrire cout au lieu de std::cout

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

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

```
class Cercle : public Figure {
};

class Carre : public Figure {
};

class Triangle : public Figure {
};
```

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

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

Plutôt que d'encapsuler la collection dans la classe, je préfère ici dire que la classe `Dessin` **EST UNE** collection, et donc hériter de la classe `vector`. Pour pouvoir bénéficier du polymorphisme, les constituants de cette collection doivent être des pointeurs.

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

```
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. [...]

```
class Cercle : public Figure {
public:
  Cercle(double x = 0.0) {
    cout << "Et hop, un cercle de plus !" << endl;
    rayon = x;
  }
  Cercle(const Cercle& c) {
    cout << "Et encore un cercle qui fait des petits !" << endl;
    rayon = c.rayon;
  }
  ~Cercle() { cout << "le dernier cercle ?" << endl; }
private:
  double rayon;
};

class Carre : public Figure {
public:
  Carre(double x = 0.0) {
    cote = x;
    cout << "Coucou, un carré de plus !" << endl;
  }
  Carre(const Carre& c) {
    cout << "Et encore un carré qui fait des petits !" << endl;
    cote = c.cote;
  }
```

```
  ~Carre() { cout << "bou... un carré de moins." << 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 arrivé !" << endl;
  }
  Triangle(const Triangle& t) {
    cout << "Et encore un triangle qui fait des petits !" << endl;
    base = t.base;
    hauteur = t.hauteur;
  }
  ~Triangle() { cout << "La fin du 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 ?

  ben .. en utilisant le constructeur de copie.

  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 coté " << 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 (unsigned int i(0); i < size(); ++i) delete (*this)[i];
}
```

Notez que puisque que la classe `Dessin` hérite de `vector` elle possède elle-même une méthode `size` et un opérateur `[]`.

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

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

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 avertit par les warnings.

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

Voici **ici le code complet**.

[...] *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 de 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.

Dernière mise à jour : $Date: 2013/04/04 17:48:17 $ (`$Revision: 1.5 $`**)**