TP POO C++ : Le polymorphisme

© 2013-2017 tv <tvaira@free.fr> - v.1.1

Notions de base 2

Notion d'héritage . . . 2

Notion de redéfinition . . . 2

Notion de polymorphisme . . . 2

Notion de fonctions virtuelles . . . 2

Les figures géométriques 3

Présentation . . . 3

Le problème . . . 3

La solution . . . 9

Travail demandé 10

Objectifs . . . 10

Présentation . . . 10

Les classes `Figure`, `Triangle` et `Carre` . . . 10

Exercice n°1 . . . 13

Classe abstraite 17

Définition . . . 17

Besoin . . . 17

Exemple . . . 17

Exercice n°2 . . . 19

UML . . . 19

Annexe : Le principe de substitution de Liskov 20

TP POO C++ : Le polymorphisme

Les objectifs de ce TP sont de découvrir la programmation orientée objet en C++ et plus particulèrement les notions de polymorphisme et de classe abstraite.

Notions de base

Notion d'héritage

L'**héritage** est un **concept fondamental de la programmation orientée objet**. Il se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.

L'héritage permet **d'ajouter des propriétés (attributs et/ou méthodes) à une classe existante pour en obtenir une nouvelle plus précise**. Il permet donc **la spécialisation ou la dérivation de types**.

Notion de redéfinition

Il ne faut pas confondre la redéfinition et la surdéfinition :
- Une **surdéfinition (ou surcharge)** permet **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente**.
- Une **redéfinition (*overriding*)** permet **de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer**. Elle doit avoir une signature rigoureusement identique à la méthode parente.

Un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire ou de la compléter.

Notion de polymorphisme

Le **polymorphisme** représente **la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation**.

On voit donc apparaître ici le concept de **polymorphisme** : **choisir en fonction des besoins quelle méthode appeler et ce au cours même de l'exécution**.

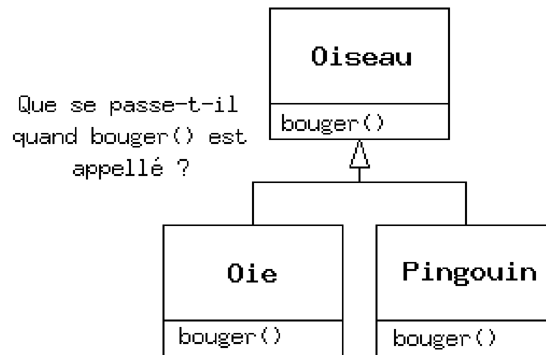
Le **polymorphisme** est implémenté **en C++** avec **les fonctions `virtual` et l'héritage**.

Notion de fonctions virtuelles

Pour créer une fonction membre `virtual`, il suffit de faire précéder la déclaration de la fonction du mot-clef `virtual`. Seule, la déclaration nécessite ce mot-clef, pas la définition. Si une fonction est déclarée `virtual` dans la classe de base, elle est `virtual` dans toutes les classes dérivées.

La redéfinition d'une fonction virtuelle dans une classe dérivée est généralement appelée **redéfinition (*overriding*)**.

Exemple : Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'`Oiseau`, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ? Car la méthode `bouger()` de la classe `Oiseau` a été déclarée `virtual`. Puis les classes `Oie` et `Pingouin` l'ont redéfinie pour obtenir le bon comportement.



Les figures géométriques

Présentation

Pour réaliser un programme orienté objet en C++ qui doit manipuler différentes figures géométriques, nous devons **développer une hiérarchie de classes**.

Le problème

Pour voir le polymorphisme en action, on va commencer par un exemple illustrant le problème lié à l'héritage si les classes ne sont pas polymorphes.

On va définir une **classe de base** pour toutes nos figures : la class `Figure`.

Figure
double x
double y
double z
+ Figure(double x = 0, double y = 0, double z = 0)
+ ~Figure()
+ string description()
+ string toString()

```
#ifndef FIGURE_H
#define FIGURE_H

#include <string>
using namespace std;

class Figure
{
   protected:
      double x, y, z; //coordonnées du centre
```

 © 2013-2017 <tvaira@free.fr> v.1.1

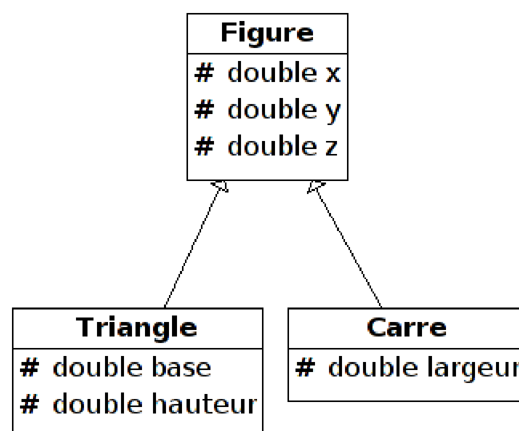
```
   public:
      Figure(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}
      ~Figure() {}
      string description() const;
      string toString() const; // la valeurs des attributs sous forme de string
};

#endif
```

Figure.h

```
#include "Figure.h"

#include <sstream>
#include <string>
using namespace std;

string Figure::description() const
{
   return "Figure"; // je suis ...
}

string Figure::toString() const
{
   ostringstream oss;
   oss << x << " " << y << " " << z; // mes attributs sous forme de string
   return oss.str();
}
```

Figure.cpp

On va maintenant définir sous forme de classe les deux figures géométriques suivantes : le `Triangle` et le `Carre`. Évidemment, le **triangle** est une **figure** et le **carré** est une **figure** aussi. On va donc mettre en oeuvre une relation d'**héritage** entre les classes `Triangle`, `Carre` et `Figure` :



Maintenant, il nous reste qu'à utiliser la **redéfinition** des méthodes `description()` et `toString()` de la classe mère `Figure`.

Rappel : Une redéfinition (overriding) permet de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer.

```
#ifndef TRIANGLE_H
```

```
#define TRIANGLE_H

#include <cmath>
#include <string>
#include "Figure.h"

using namespace std;

class Triangle : public Figure
{
   protected:
      double base, hauteur; // les propriétés spécifiques d'un Triangle

   public:
      Triangle(double base, double hauteur, double x = 0, double y = 0)
      : Figure(x, y, 0), base(base), hauteur(hauteur) {}
      ~Triangle() {}
      string description() const; // redéfinition pour un Triangle
      string toString() const; // redéfinition pour un Triangle
};

#endif
```

Triangle.h

```
#include "Triangle.h"

#include <sstream>
#include <string>
using namespace std;

string Triangle::description() const
{
   return "Triangle"; // je suis plus précisément ...
}

string Triangle::toString() const
{
   ostringstream oss;
   oss << base << " " << hauteur; // mes attributs spécifiques
   return Figure::toString() + " " + oss.str();
}
```

Triangle.cpp

Une classe dérivée (ici la classe `Triangle`) peut appeler une méthode de la classe de base (ici `Figure`) de la manière suivante : `Figure::toString()`

```
#ifndef CARRE_H
#define CARRE_H

#include <cmath>
#include <string>
```

 © 2013-2017 <tvaira@free.fr> v.1.1

```
#include "Figure.h"

using namespace std;

class Carre : public Figure
{
   protected:
      double largeur; // une propriété spécifique aux carrés

   public:
      Carre(double largeur, double x = 0, double y = 0)
      : Figure(x, y, 0), largeur(largeur) {}
      ~Carre() {}
      string description() const; // redéfinition pour un Carre
      string toString() const; // redéfinition pour un Carre
};

#endif
```

Carre.h

```
#include "Carre.h"

#include <sstream>
#include <string>
using namespace std;

string Carre::description() const
{
   return "Carré"; // je suis plus précisément ...
}

string Carre::toString() const
{
   ostringstream oss;
   oss << largeur; // moi, j'ai une largeur en plus des attributs d'une Figure
   return Figure::toString() + " " + oss.str();
}
```

Carre.cpp

On peut écrire un programme de test où l'on va instancier un triangle et un carré.

```
#include <iostream>

using namespace std;

#include "Figure.h"
#include "Triangle.h"
#include "Carre.h"

int main()
{
   Triangle triangle(5,8);
   Carre carre(4);
```

```
   cout << "Je suis un " << triangle.description() << endl;
   cout << "Mes attributs sont : " << " " << triangle.toString() << endl;

   cout << "Je suis un " << carre.description() << endl;
   cout << "Mes attributs sont : " << " " << carre.toString() << endl;
   cout << endl;

   return 0;
}
```

testFigures.cpp

On obtient l'affichage suivant :

```
Je suis un Triangle
Mes attributs sont : 0 0 0 5 8
Je suis un Carré
Mes attributs sont : 0 0 0 4
```

Ok ! On pourrait penser que tout fonctionne parfaitement. On va écrire le programme ci-dessous qui va illustrer des problèmes liés à l'héritage si nos classes ne sont pas polymorphes.

Problème n°1 : On commence à posséder de **nombreux types de figures**. On va introduire une fonction qui leur permettra d'afficher leur description :

```
// une fonction qui permet d'afficher la description d'une Figure
void afficherInfos(const Figure &f)
{
   cout << "Je suis un " << f.description() << endl;
   cout << "Mes attributs sont : " << " " << f.toString() << endl;
}

// Pas de problème pour les appels :
afficherInfos(triangle); // pas de souci : un triangle est une figure
afficherInfos(carre); // pas de souci : un carré est une figure
```

afficherInfos()

Problème n°2 : On commence à posséder de **nombreux objets figures**. On souhaite les conserver dans un conteneur (un `vector` ou une `list` par exemple) puis de les utiliser pour afficher leur description :

```
list<Figure *> liste; // une liste de pointeurs sur des Figure
list<Figure *>::iterator iL;
Figure *figure; // un pointeur sur une Figure

// Pas de problème pour le stockage :
liste.push_back(&triangle); // pas de souci : un triangle est une figure
liste.push_back(&carre); // pas de souci : un carré est une figure

// parcourons la liste de Figure
for (iL = liste.begin(); iL != liste.end(); iL++)
{
   figure = *iL; // récupère la figure
   cout << "Je suis un " << figure->description() << endl;
   cout << "Mes attributs sont : " << " " << figure->toString() << endl;
}
```

afficherInfos()

On écrit donc un deuxième programme de test où l'on va instancier un triangle et un carré puis utiliser ces deux figures.

```
#include <iostream>
#include <exception>
#include <iomanip>
#include <list>
#include <fstream>

using namespace std;

#include "Figure.h"
#include "Triangle.h"
#include "Carre.h"

void afficherInfos(const Figure &f);

int main()
{
   Triangle      triangle(5,8);
   Carre         carre(5);
   Figure        *figure;
   list<Figure *> liste;
   list<Figure *>::iterator iL;

   cout << endl;
   //cout << "Triangle :" << endl;
   cout << "Je suis un " << triangle.description() << endl;
   cout << "Mes attributs sont : " << " " << triangle.toString() << endl;

   //cout << "Carre :" << endl;
   cout << "Je suis un " << carre.description() << endl;
   cout << "Mes attributs sont : " << " " << carre.toString() << endl;
   cout << endl;

   // Problème n˚1 :
   cout << "Problème n°1 :" << endl;
   afficherInfos(triangle);
   afficherInfos(carre);
   cout << endl;

   // Problème n˚2 :
   cout << "Problème n°2 :" << endl;
   liste.push_back(&triangle);
   liste.push_back(&carre);

   for (iL = liste.begin(); iL != liste.end(); iL++)
   {
      figure = *iL;
      cout << "Je suis un " << figure->description() << endl;
      cout << "Mes attributs sont : " << " " << figure->toString() << endl;
   }

   cout << endl;
```

```
   return 0;
}

void afficherInfos(const Figure &f)
{
   cout << "Je suis un " << f.description() << endl;
   cout << "Mes attributs sont : " << " " << f.toString() << endl;
}
```

testFigures-2.cpp

Il n'y a pas d'erreurs à la compilation puisque un **triangle** et un **carré** sont des **figures**.

Par contre, on n'obtient pas ce que l'on désire à l'exécution :

```
Je suis un Triangle
Mes attributs sont : 0 0 0 5 8
Je suis un Carré
Mes attributs sont : 0 0 0 4

Problème n°1 :
Je suis un Figure
Mes attributs sont : 0 0 0
Je suis un Figure
Mes attributs sont : 0 0 0

Problème n°2 :
Je suis un Figure
Mes attributs sont : 0 0 0
Je suis un Figure
Mes attributs sont : 0 0 0
```

En effet, c'est la méthode `description()` de la classe `Figure` qui est appelée et non celle des classes `Triangle` et `Carre`. Même chose pour la méthode `toString()`.

La solution

Pour corriger cela, il suffit de déclarer les méthodes `description()` et `toString()` comme **virtuelles** **(`virtual`)** dans la classe `Figure` :

```
class Figure
{
   ...
   virtual void description() const;
   virtual void toString() const;
   ...
};
```

Figure.h

On obtient maintenant un **comportement polymorphe** : la « bonne méthode » `description()` est appelée en fonction du type de l'objet à l'exécution. Même chose pour la méthode `toString()`.

```
Je suis un Triangle
Mes attributs sont : 0 0 0 5 8
Je suis un Carré
```

 © 2013-2017 <tvaira@free.fr> v.1.1

```
Mes attributs sont : 0 0 0 4

Problème n°1 corrigé :
Je suis un Triangle
Mes attributs sont : 0 0 0 5 8
Je suis un Carré
Mes attributs sont : 0 0 0 4

Problème n°2 corrigé :
Je suis un Triangle
Mes attributs sont : 0 0 0 5 8
Je suis un Carré
Mes attributs sont : 0 0 0 4
```

Travail demandé

Objectifs

On désire réaliser un programme orienté objet en C++ qui **fournit une hiérarchie de classe destinées à mémoriser ou manipuler les propriétés de différentes figures géométriques**.

Présentation

On propose de traiter au moins le cas des **rectangles, cercles, triangles, carrés, sphères, parallélépipèdes rectangles, cubes, ...** mais cette liste n'est pas limitative.

Chaque classe devra permettre de mémoriser les données qui permettent de définir une instance, par exemple la longueur des deux cotés d'un rectangle, le rayon de la sphère, etc. Chaque instance devra disposer quand cela a un sens :
- d'une méthode `double perimetre()` qui renvoie la valeur du périmètre de l'instance sur laquelle on l'appelle ;
- d'une méthode `double aire()` qui renvoie l'aire de la surface de l'objet concerné ;
- d'une méthode `double volume()` qui renvoie le volume de la forme concernée.

On considérera que lorsque ces trois notions ne sont pas définies mathématiquement, alors la valeur renvoyée sera **nulle**. Par exemple, le volume d'un carré sera nul, et le périmètre d'une sphère également.

Les classes `Figure`, `Triangle` et `Carre`

On vous fournit les classes `Figure`, `Triangle` et `Carre`.

```
typedef enum {courte, longue} t_desc;

class Figure
{
   protected:
      double x, y, z; //coordonnées du centre

   public:
```

```
      Figure(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}
      virtual ~Figure() {}
      virtual double perimetre() { return 0; }
      virtual double aire()     { return 0; }
      virtual double volume()   { return 0; }
      virtual string description(t_desc type) { return "Figure"; }
      virtual string toString();
};

string Figure::toString()
{
   ostringstream oss;
   oss << x << " " << y << " " << z;
   return oss.str();
}
```

Figure

```
class Triangle : public Figure
{
   protected:
      double base, hauteur;

   public:
      Triangle(double base, double hauteur, double x = 0, double y = 0)
      : Figure(x, y, 0), base(base), hauteur(hauteur) {}
      ~Triangle() {}
      double perimetre() { return sqrt(base * base + hauteur * hauteur) + base + hauteur; }
      double aire()      { return base * hauteur / 2; }
      string description(t_desc type);
      string toString();
};

string Triangle::description(t_desc type)
{
   string desc;
   if(type == courte)
   {
      desc = "Triangle";
   }
   else if(type == longue)
   {
      ostringstream oss;
      desc = Figure::description(type) + " <|- Triangle\n";
      desc += "{\n";
      desc += "\tx = "; oss << x; desc += oss.str() + "\n";
      desc += "\ty = "; oss.str(""); oss << y; desc += oss.str() + "\n";
      desc += "\tz = "; oss.str(""); oss << z; desc += oss.str() + "\n";
      desc += "\tbase = "; oss.str(""); oss << base; desc += oss.str() + "\n";
      desc += "\thauteur = "; oss.str(""); oss << hauteur; desc += oss.str() + "\n";
      desc += "}\n";
   }
   else
   {
```

 © 2013-2017 <tvaira@free.fr> v.1.1

```
      desc = "";
   }
   return desc;
}

string Triangle::toString()
{
   ostringstream oss;
   oss << base << " " << hauteur;
   return Figure::toString() + " " + oss.str();
}
```

Triangle

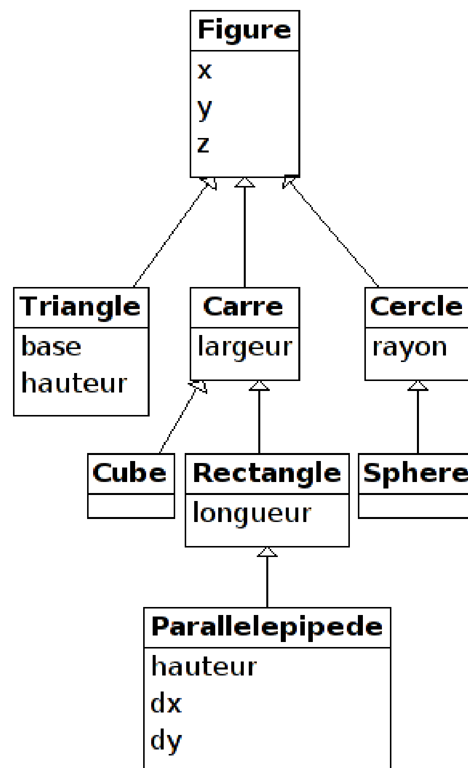
```
class Carre : public Figure
{
   protected:
      double largeur;

   public:
      Carre(double largeur, double x = 0, double y = 0)
      : Figure(x, y, 0), largeur(largeur) {}
      ~Carre() {}
      double perimetre() { return largeur * 4; }
      double aire()      { return largeur * largeur; }
      string description(t_desc type);
      string toString();
};

string Carre::description(t_desc type)
{
   string desc;
   if(type == courte)
   {
      desc = "Carré";
   }
   else if(type == longue)
   {
      ostringstream oss;
      desc = Figure::description(type) + " <|- Carre\n";
      desc += "{\n";
      desc += "\tx = "; oss << x; desc += oss.str() + "\n";
      desc += "\ty = "; oss.str(""); oss << y; desc += oss.str() + "\n";
      desc += "\tz = "; oss.str(""); oss << z; desc += oss.str() + "\n";
      desc += "\tlargeur = "; oss.str(""); oss << largeur; desc += oss.str() + "\n";
      desc += "}\n";
   }
   else
   {
      desc = "";
   }
   return desc;
}
```

```
string Carre::toString()
{
   ostringstream oss;
   oss << largeur;
   return Figure::toString() + " " + oss.str();
}
```

Carre

Exercice n°1

On vous demande d'écrire les classes `Cercle`, `Cube`, `Parallelepipede`, `Rectangle` et `Sphere` et de faire fonctionner le programme de test fourni.



Mathématiquement, un carré est un rectangle et un héritage aurait pu être implémenté entre les deux classes. Mais cela pose un problème (cf. violation du principe de substitution de Liskov en Annexe) qui dépasse le cadre de ce TP.

```
#include <iostream>
#include <iomanip>
#include <list>
#include <fstream>

using namespace std;

#include "Figure.h"
#include "Triangle.h"
#include "Carre.h"
#include "Cercle.h"
```

```
#include "Cube.h"
#include "Parallelepipede.h"
#include "Rectangle.h"
#include "Sphere.h"

int main()
{
   cout << endl;

   Triangle      triangle(5,8); // base, hauteur
   //cout << "Triangle :" << endl;
   cout << "Je suis un " << triangle.description(courte) << endl;
   cout << triangle.description(longue);
   cout << "Perimetre = " << triangle.perimetre() << endl;
   cout << "Aire = "    << triangle.aire() << endl;
   cout << "Volume = "  << triangle.volume() << endl << endl;

   Carre         carre(4); // largeur
   //cout << "Carre :" << endl;
   cout << "Je suis un " << carre.description(courte) << endl;
   cout << carre.description(longue);
   cout << "Périmetre = " << carre.perimetre() << endl;
   cout << "Aire = "    << carre.aire() << endl;
   cout << "Volume = "  << carre.volume() << endl << endl;

   Rectangle     rectangle(5, 4); // largeur, longueur
   //cout << "Rectangle : " << endl;
   cout << "Périmetre = " << rectangle.perimetre() << endl;
   cout << "Aire = "    << rectangle.aire() << endl;
   cout << "Volume = "  << rectangle.volume() << endl << endl;

   Cercle        cercle(2); // rayon
   //cout << "Cercle : " << endl;
   cout << "Périmetre = " << cercle.perimetre() << endl;
   cout << "Aire = "    << cercle.aire() << endl;
   cout << "Volume = "  << cercle.volume() << endl << endl;

   Cube          cube(6); // largeur
   //cout << "Cube : " << endl;
   cout << "Périmetre = " << cube.perimetre() << endl;
   cout << "Aire = "    << cube.aire() << endl;
   cout << "Volume = "  << cube.volume() << endl << endl;

   Parallelepipede parallelepipede(5, 4, 9); // largeur, longueur, hauteur
   //cout << "Parallelepipede : " << endl;
   cout << "Périmetre = " << parallelepipede.perimetre() << endl;
   cout << "Aire = "    << parallelepipede.aire() << endl;
   cout << "Volume = "  << parallelepipede.volume() << endl << endl;

   Sphere        sphere(3); // rayon
   //cout << "Sphere : " << endl;
   cout << "Périmetre = " << sphere.perimetre() << endl;
   cout << "Aire = "    << sphere.aire() << endl;
```

```
cout << "Volume = "  << sphere.volume() << endl << endl;

Figure         *fig;
list<Figure *> liste;
list<Figure *>::iterator iL;
fstream *fichier = NULL;
string figure;
int    x, y, z;
double base, hauteur, largeur;

liste.push_back(&triangle);
liste.push_back(&carre);
liste.push_back(&rectangle);
liste.push_back(&cercle);
liste.push_back(&cube);
liste.push_back(&parallelepipede);
liste.push_back(&sphere);

/*
// Test du polymorphisme :
for (iL = liste.begin(); iL != liste.end(); iL++) {
   fig = *iL;
   cout << fig->description(courte) << " " << fig->toString() << endl;
   cout << fig->description(longue) << endl;
   cout << "Périmetre = " << fig->perimetre() << endl;
   cout << "Aire = "    << fig->aire() << endl;
   cout << "Volume = "  << fig->volume() << endl << endl;
}*/

fichier = new fstream("Figures.txt", fstream::out);
for (iL = liste.begin(); iL != liste.end(); iL++)
{
   fig = *iL;
   *fichier << fig->description(courte) << " " << fig->toString() << endl;
}
fichier->close();
delete fichier;

fichier = new fstream("Figures.txt", fstream::in);
*fichier >> figure;
while (! fichier->eof())
{
   *fichier >> x >> y >> z;
   cout << figure;
   cout << " " << x << " " << y << " " << z;
   if (figure == "Figure")
      cout << endl;
   if (figure == "Triangle")
   {
      *fichier >> base >> hauteur;
      cout << " " << base << " " << hauteur << endl;
   }
   if (figure == "Carré")
```

 © 2013-2017 <tvaira@free.fr> v.1.1

```
      {
         *fichier >> largeur;
         cout << " " << largeur << endl;
      }
      if (figure == "Rectangle")
      {
         *fichier >> largeur >> longueur;
         cout << largeur << longueur << endl;
      }
      if (figure == "Cercle")
      {
         *fichier >> rayon;
         cout << rayon << endl;
      }
      if (figure == "Cube")
      {
         *fichier >> largeur;
         cout << largeur << endl;
      }
      if (figure == "Parallélépipède")
      {
         *fichier >> largeur >> longueur >> hauteur >> dx >> dy;
         cout << largeur << longueur << hauteur << dx << dy << endl;
      }
      if (figure == "Sphère")
      {
         *fichier >> rayon ;
         cout << rayon << endl;
      }
      while (fichier->get() != '\n');
      *fichier >> figure;
   }
   fichier->close();
   delete fichier;

   return 0;
}
```

testFigures.cpp

Pour information, voici les résultats obtenus pour les calculs des périmètres, aires et volumes des différentes figures :

```
Triangle :
Périmetre = 22.434
Aire = 20
Volume = 0

Carre :
Périmetre = 16
Aire = 16
Volume = 0

Rectangle :
Périmetre = 18
```

```
Aire = 20
Volume = 0

Cercle :
Périmetre = 12.5664
Aire = 12.5664
Volume = 0

Cube :
Périmetre = 0
Aire = 216
Volume = 216

Parallelepipede :
Périmetre = 0
Aire = 202
Volume = 180

Sphere :
Périmetre = 0
Aire = 113.097
Volume = 113.097
```

Décommenter ensuite la partie "Test du polymorphisme" et vérifier que vos classes sont bien polymorphes. Expliquer.

TP POO C++ : Les classes abstraites

Classe abstraite

Définition

En programmation orientée objet (POO), une **classe abstraite** est une classe dont l'implémentation n'est pas complète et qui n'est donc pas instanciable (on ne peut pas créer d'objet à partir de cette classe).

Besoin

Une **classe abstraite** sert de base à d'autres classes dérivées (héritées). Le mécanisme des classes abstraites permet de définir des comportements (méthodes) qui devront être implémentés dans les classes filles, mais sans fournir elle-même ces comportements (c'est-à-dire sans écrire de code pour cette méthode). Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite. Ce contrat est une interface de programmation.

Exemple

En C++, une classe est abstraite si elle contient au moins une **méthode déclarée virtuelle pure**, c'est-à-dire commençant par `virtual` et terminée par `= 0`. Ce type de classe n'est pas instanciable. Une

fonction virtuelle pure doit être définie ou redéclarée explicitement virtuelle pure dans les classes dérivées.

On peut décider que la classe `Figure` de notre exemple soit une **classe abstraite**. Pour cela, on va déclarer les méthodes `perimetre()`, `volume()` et `aire()` comme des **fonctions virtuelles pures**.

```
class Figure
{
   protected:
      double x, y, z; // coordonnées du centre

   public:
      Figure(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}
      virtual ~Figure() {}
      // TODO ...
      virtual double volume() = 0; // méthode virtuelle pure
      virtual string description(t_desc type) {return "Figure";}
      virtual string toString();
};
```

La classe Figure est maintenant une classe abstraite

On ne peut plus instancier d'objets de type `Figure` ! D'autre part, l'ensemble des classes qui héritent de la classe Figure devront définir les méthodes `perimetre()`, `volume()` et `aire()`. Ce sont des aspects vérifiés par le compilateur.

```
Figure f; // essayons d'instancier un objet f de type Figure
```

testFigure.cpp

Erreur n°1 : En essayant d'instancier un objet de la classe Figure, on obtient une erreur à la compilation (les messages de `g++` sont suffisamment clairs et précis) :

```
// essayons de compiler :
$ g++ -c testFigure.cpp
testFigure.cpp: In function 'int main()':
testFigure.cpp: erreur: cannot declare variable 'f' to be of abstract type 'Figure'
In file included from testFigure.cpp:
Figure.h: note: because the following virtual functions are pure within 'Figure':
Figure.h: note: virtual double Figure::perimetre()
Figure.h: note: virtual double Figure::aire()
Figure.h: note: virtual double Figure::volume()
```

Erreur n°2 : Si la classe `Triangle` (les classes `Carre` et `Cercle` aussi ...) n'implémente pas la méthode `volume()`, on obtient une erreur à la compilation (les messages de `g++` sont suffisamment clairs et précis) :

```
// essayons de compiler :
$ g++ -c testFigure.cpp
testFigure.cpp: In function 'int main()':
testFigure.cpp: erreur: cannot declare variable 'triangle' to be of abstract type 'Triangle'
In file included from testFigure.cpp:
Triangle.h: note: because the following virtual functions are pure within 'Triangle':
Triangle.h: note: virtual double Figure::volume()
```

Le compilateur considère que la classe `Triangle` est elle-même abstraite car la méthode `volume()` n'est pas définie.

Pour corriger cette erreur, il suffira de définir la méthode `volume()` dans la classe `Triangle` (dans `Carre` et `Cercle` aussi ...) :

```
class Triangle : public Figure
{
   protected:
      double base, hauteur;

   public:
      Triangle(double base, double hauteur, double x = 0, double y = 0) : Figure(x, y, 0),
         base(base), hauteur(hauteur) {}
      ~Triangle() {}
      double perimetre() {return sqrt(base * base + hauteur * hauteur) + base + hauteur;}
      double volume()    {return 0;} // je dois définir la méthode volume()
      double aire()      {return base * hauteur / 2;}
      string description(t_desc type);
      string toString();
};
```

La classe Triangle corrigée

Exercice n°2

La classe Figure est abstraite et doit posséder trois méthodes virtuelles pures : `perimetre()`, `volume()` et `aire()`.

On vous demande de modifier les classes `Figure` et ses dérivées afin de faire fonctionner le programme de test fourni précedemment.

UML

En C++, une classe abstraite sera indiquée par son nom écrit en ***italique*** ainsi que toutes ses méthodes virtuelles pures :

Figure
x y z
Figure() ~Figure() *perimetre()* *aire()* *volume()* description() toString()

 © 2013-2017 <tvaira@free.fr> v.1.1

Annexe : Le principe de substitution de Liskov

Le principe de substitution de Liskov (LSP) est, en programmation orientée objet, une définition particulière de la notion de sous-type. Il a été formulé par Barbara Liskov et Jeannette Wing dans un article intitulé "*Family Values : A Behavioral Notion of Subtyping*".

Liskov et Wing en ont proposé la formulation condensée suivante : Si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T.

L'exemple classique d'une **violation du LSP** est la suivante :

- Soit une classe Rectangle représentant les propriétés d'un rectangle : hauteur, largeur. On lui associe donc des accesseurs pour accéder et modifier la hauteur et la largeur librement. On défini la règle "hauteur" et "largeur" sont librement modifiable.
- Soit une classe Carré que l'on fait dériver de la classe Rectangle. En effet, en mathématique, un carré est un rectangle. Donc, on défini naturellement la classe Carré comme sous-type de la classe Rectangle. On défini la règle "les 4 cotés du carré doivent être égaux".

On s'attend de pouvoir utiliser une instance de type Carré n'importe où un type Rectangle est attendu.

Problème : Un carré ayant par définition quatre cotés égaux, il convient de restreindre la modification de la hauteur et de la largeur pour qu'elles soient toujours égales. Néanmoins, si un carré est utilisé là où, comportementalement, on s'attend à interagir avec un rectangle, des comportements incohérents peuvent subvenir : Les cotés d'un carré ne peuvent être changés indépendamment, contrairement à un rectangle. Une mauvaise solution consisterait à modifier les *setter* du carré pour préserver l'invariance de ce dernier. Mais ceci violerait la règle des *setter* du rectangle qui spécifie que l'on puisse modifier hauteur et largeur indépendamment.

La solution consiste à ne pas considérer un type Carré comme substitut d'un type Rectangle, et les définir comme deux types complètement indépendent. Ceci ne contredit pas le fait qu'un carré soit un rectangle. La classe Carré est un représentant du concept "carré". La classe Rectangle est un représentant du concept "rectangle". Or, les représentants ne partagent pas les mêmes propriétés que ce qu'ils représentent.