Langage C/C++

TD 5 : programmation orientée objet

Hubert Godfroy

11 décembre 2014

1 Préambule

On utilisera la classes `Base` :

```
class Base {
public:
    Base();
      void afficher();
      void initial();
};
Base::Base() {
    cout << "construction d'une instence de Base\n";
}
void Base::afficher(){
    cout << "fonction afficher() dans la classe Base\n";
}
void Base::initial(){
    cout << "fonction initial() dans la classe Base\n";
}
```

et la sous-classe `Derive` héritant de `Base`

```
class Derive : public Base
{
public:
    Derive();
    Derive (int n);
    void initial();
    void afficher();
};
Derive::Derive(){
    cout << "construction d'une instance de Derive\n";
}
Derive::Derive(int n){
    cout << "construction d'une instance de Derive(int n)\n";
}
```

```
15 void Derive::afficher(){
16     cout << "fonction afficher() dans la classe Derive\n";
17 }
18 void Derive::initial(){
19     cout << "fonction initial() dans la classe Derive\n";
20 }
```

2 Liaisons

2.1 Généralités

Question 1 :

Exécuter le programme suivant :

```
1 Derive* test;
2 test = new Derive();
3 test->afficher();
4 test->initial();
```

Observer la sortie du programme. Changer le type de `test` en `Base*` (uniquement dans la première ligne). Qu'est-ce que cela change sur la sortie du programme ? Pourquoi ?

Passer la méthode `afficher()` en virtuelle dans la déclaration de la classe `Base`. Qu'est-ce que cela change sur la sortie du programme ? Qu'en déduisez vous sur la sémantique d'une fonction virtuelle ?

Question 2 :

Reprendre le programme du slide 23 du cours (le programme JAVA). Pourquoi est-il impossible de savoir à l'avance quelle méthode `afficher` sera appelée ?

Convertir le programme JAVA en C++ de telle sorte qu'il ait la ***même*** sémantique au niveau des sorties. Écrire le même programme en C.

Comment en C++ peut on s'affranchir cette indécidabilité ? Le faire…

Écrire un programme en C équivalent à ce dernier programme.

2.2 Destructeurs

Question 3 :

Surcharger les destructeurs par défaut des classes `Base` et `Derive`. En vous aidant des questions précédentes, prévoir le comportement de la fonction `delete` dans le programme suivant :

```
Base* test = new Derive();
delete test;
```

Comment résoudre le problème ? Le faire…

3 Objets statiques/dynamiques

Question 4 :
Surcharger les constructeurs par défauts de `Base` `Derive` avec

```
Base::~Base(){cout << "destruction de Base\n";}
Derive::~Derive(){cout << "destruction de Derive\n";}
```

Observer le comportement du programme

```
Derive obj;
```

puis du programme

```
Derive* obj = new Derive();
```

Comparer ces observations avec le comportement de C.

4 Spécification de programme

Dans cette section on souhaite dans un premier temps spécifier les caractéristiques de différents types d'objets géométriques, puis instancier ces définitions.

Question 5 :

(a) Définir une classe abstraite pure `Polygone` disposant des méthodes `périmètre` et `aire`.

(b) Définir une classe `Triangle` héritant de cette classe.

(c) Définir une sous-classe `Parallélogramme` de `Polygone`.

Question 6 :

(a) Définir une classe `Couleur`.

(b) Définir une classe abstraite pure `Coloré` déclarant une méthode `couleur` renvoyant une couleur de type `Couleur`.

Question 7 :
Définir une classe `QuadrilatèreColoré` implémentant les classes abstraites `Polygone` et `Coloré`.

5 Interfaces graphiques

Dans cette section on partira d'un projet Qt vide. On ajoutera dans le fichier `<nomdupprojet>.pro` la ligne

```
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

On utilisera le programme de base suivant :

```
#include <QApplication>
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    ...
    return app.exec();
}
```

Question 8 :

(a) En utilisant la classe `QWidget`, créer une fenêtre. Ne pas oublier d'appeler la méthode `show` sur la fenêtre à la fin de sa création.

(b) Redimentionner la fenêtre avec la méthode `setFixedSize(x, y)`.

(c) Ajouter un bouton à la fenêtre.

```
QPushButton bouton("plop", &fen);
```

(d) Déplacer le bouton avec la méthode `move(x, y)`.

Question 9 :

Afin de personnaliser le comportement de la fenêtre, créer une classe `Fenêtre` héritant de `QWidget` (spécifier la classe mère directement dans la fenêtre de dialogue de QtCreator ouverte lors de la demande de création de classe) en lui ajoutant un champs de type `QPushButton*` qui sera initialisé dans le constructeur de la fenêtre.

Question 10 :

Construire un destructeur pour cette classe.

Question 11 :

Ajouter la méthode `void on_click(void)` changeant le texte du bouton (utiliser la méthode `setText("nom")`).

Question 12 :

Ajouter un label (classe `QLabel`) et un slider (classe `QSlider`) à la fenêtre (ne pas oublier d'adapter les constructeurs et destructeurs).

Question 13 :

Connecter le label au slider de telle sorte que le label indique la valeur du slider (utiliser la méthode `valueChanged`) du slider.

Question 14 :

Ajouter les champs `QGraphicsView* view` et `QGraphicsScene* scene`.

Question 15 :

Initialiser le champ `scene` avec le constructeur `QGraphicsScene(`x_1`, `y_1`, `x_2`, `y_2`, parent)` et le champs `vue` avec le constructeur `QGraphicsView(scene, parent)`.

Question 16 :

Ajouter un bouton qui, lorsqu'il est cliqué, ajoute une ligne dans la scène (utiliser la méthode `addLine(`x_1`, `y_1`, `x_2`, `y_2`))`