Lecture 15

Object-Oriented Programming XI

Pure Virtual Functions

Prof. Hyeong-Seok Ko
Seoul National University
Graphics & Media Lab

Contents

- Pure virtual functions (15.6)
- Abstract classes (15.6)
- Example of an abstract class

Pure Virtual Functions

- A pure virtual function is defined by writing =0 after the function parameter list.

- Defining a function as pure virtual indicates that the function provides only the interface so that the derived classes must override the null definition.
  - The pure virtual function must be implemented by the derived class. Otherwise, it creates a compilation error.

```
class Base {
public :
   virtual void func() = 0;    // pure virtual function
};
```

Abstract Class

- A class containing one or more pure virtual functions.

```
class Base {                                    // abstract class
public :
   virtual void func() = 0;
};

class Derived : public Base {                   // abstract class
};

class Derived_2 : public Derived {              // not abstract class
public :
   void func() {
      std::cout << "func() in Derived_2" << std::endl;
   }
};
```

Instantiation of an Abstract Class

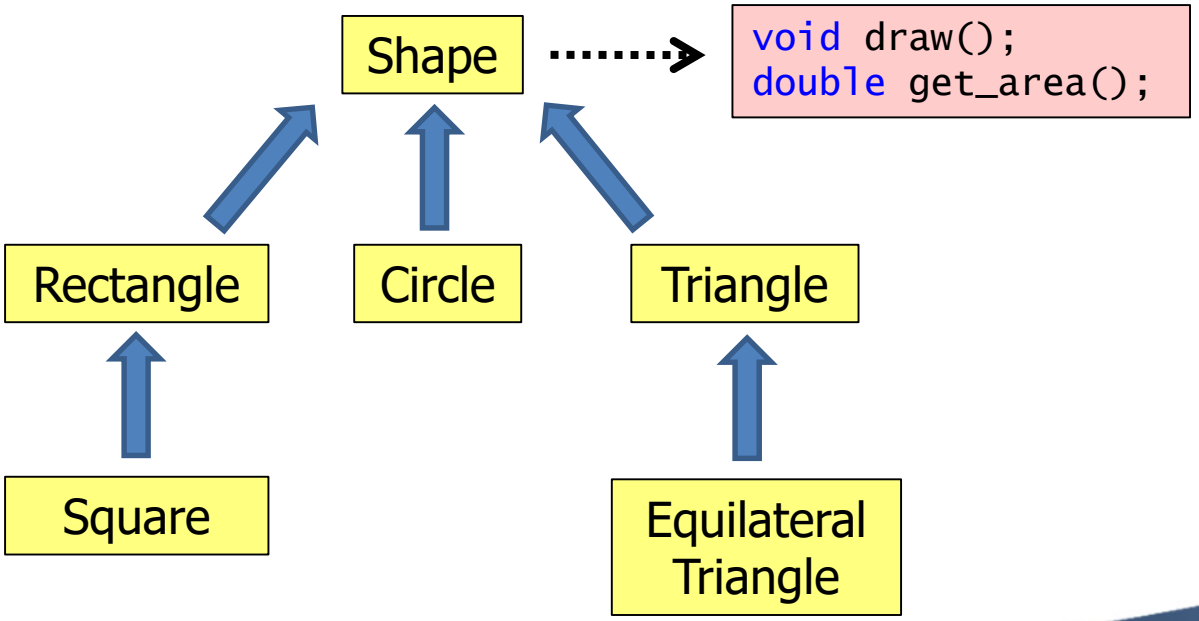
- Instantiation of an abstract class causes a compilation error.

```
class Base {                                // abstract class
public :
   virtual void func() = 0;
};

class Derived : public Base {               // abstract class
};

class Derived_2 : public Derived {          // not abstract class
public :
   void func() {
      std::cout << "func() in Derived_2" << std::endl;
   }
};

void main() {
   Base base;                               // Compilation Error
   Derived derived;                         // Compilation Error

   Base * ptr_1 = new Base();               // Compilation Error
   Base * ptr_2 = new Derived();            // Compilation Error

   // cannot instantiate abstract class
   // due to 'void Base::func()' :
}
```

Example of an Abstract Class (Shape)

- A pure virtual function provides an **interface** for the derived classes to override.
- Actual implementations should be made by the derived classes

```
class Shape {                                   // abstract base class
public :
   virtual void draw() = 0;
   virtual double get_area() = 0;
};
```



Container Using Abstract Class

```
class Shape {                          // abstract base class
public :
   virtual void draw() = 0;
};

class Rectangle : public Shape {
public :
   void draw() { std::cout << "Draw Rectangle" << std::endl; }
};

class Square : public Shape {
public :
   void draw() { std::cout << "Draw Square" << std::endl;    }
};

class Triangle : public Shape {
public :
   void draw() { std::cout << "Draw Triangle" << std::endl;  }
};

void main() {
   std::vector<Shape*> shapes;
   shapes.push_back(new Rectangle());
   shapes.push_back(new Square());
   shapes.push_back(new Triangle());

   for(std::vector<Shape*>::iterator it=shapes.begin();it!=shapes.end();++it)
      (*it)->draw();
}
```