LAB11
Polymorphism and Virtual Functions

DEE 1319

Department of Electronics Engineering

National Chiao Tung University

Learning Objectives

- Virtual functions
- Polymorphism
- Abstract classes and pure virtual functions

Virtual vs. Non-Virtual Functions (1/2)

- For non-virtual (member) functions
  - Function calls are **STATICALLY bound** (i.e., bound at compile time)

```
class B {
    public: void mf();
};
```

```
class D:public B {
    public: void mf(); //redefine mf();
}
```

```
void f() {
    B b, *pB= &b; D d, *pD= &d;
    b.mf();     // statically binding ➔ b is of type B ➔ call B::mf()
    d.mf();     // statically binding ➔ d is of type D ➔ call D::mf()
    pB->mf(); // statically binding ➔ pB is of type B* ➔ call B::mf()
    pD->mf(); // statically binding ➔ pD is of type D* ➔ call D::mf()
    pB= &d;   // ok, D is derived from B
    pB->mf(); // still statically binding ➔ pB is of type B* ➔ call B::mf()
}               // though pB actually points to d (an object of type D)
```

Virtual vs. Non-Virtual Functions (2/2)

- For virtual (member) functions
  - Must be non-static member functions
  - Function calls are **DYNAMICALLY bound** (i.e., bound at runtime) if they are invoked through pointers or references

```
class B {
    public: virtual  void mf();
};

class D:public B {
    public: void mf(); //override mf();
}
```

```
void f() {
    B b, *pB= &b; D d, *pD= &d;
    b.mf();  d.mf();
    pB->mf();  pD->mf();
    pB= &d;  // ok, D is derived from B
    pB->mf();// dynamically binding ➔ pB actually points to d ➔ call D::mf()
}
```

Object slicing vs. Virtual function

- Manipulate objects through pointers if you want to use virtual functions

```
void f() {
    B b, *pB= &b; D d, *pD= &d;

     pB= &d;
     pB->mf();// call D::mf()

     b = d; // ok, upcasting
     b.mf(); // object slicing!!   Always call B::mf()
}
```

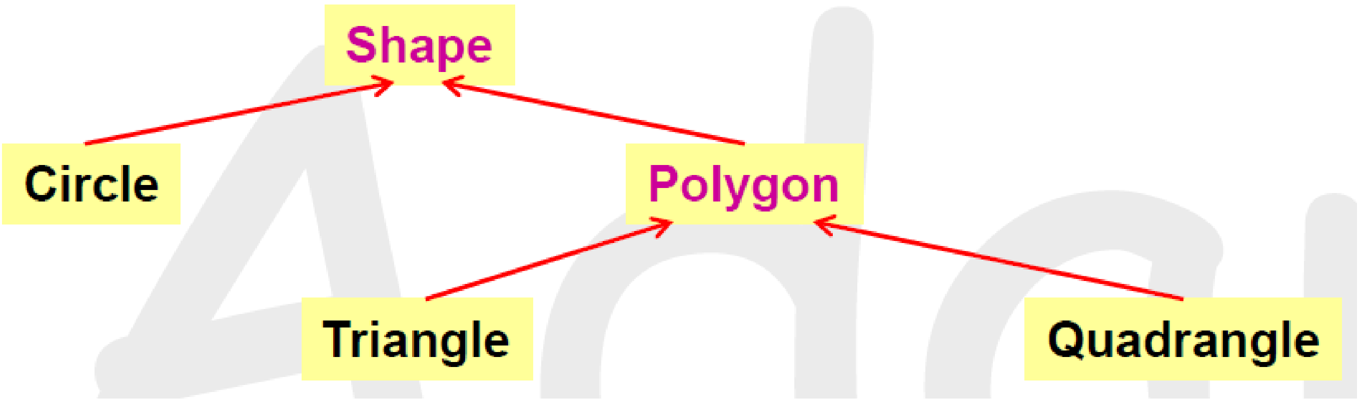
```
 class B {
     public: virtual  void mf();
 };

class D:public B {
    public: void mf(); //override mf();
}
```

Polymorphism

- Polymorphism
  - While accessing a member function, the correct version based on the actual calling object is always invoked
  - Namely, the behavior of calling a member function through a pointer/reference may be different ➔ polymorphic
- In C++, polymorphism is achieved through
  - Virtual functions, and
  - Manipulating objects through pointers or references
- A class with virtual functions is called a polymorphic class
- Polymorphism is another cornerstone of OOP

Concrete Class vs. Abstract Class

- Some concepts are concrete and some are abstract



- Abstract classes: Shape and Polygon
  - e.g., no idea how to draw or rotate an arbitrary shape
  - Objects of abstract classes should not exist (they are abstract)
- Concrete classes : Circle, Triangle and Quadrangle
  - Objects of these types can exist
  - They can be drawn, rotated, …

Pure Virtual Functions & Abstract Class

```
class Shape {
    public:
    virtual void rotate(int) = 0;  // pure virtual function
    virtual void draw() = 0;       // pure virtual function
    virtual boolis_closed() = 0; // pure virtual function
    // …                          // only declaration; no definition
};
void f() {
    Shape s; // compilation error! it must be an error, or
    s.draw(); //would be legal ; but draw() is a pure virtual function
};
```

- A class with one or more pure virtual functions is called an abstract class
- No objects of abstract class can be created in C++

Abstract Base Class (ABC)

- Abstract class is always used as a base class ➔(ABC)
  - You cannot create objects of abstract class
  - It only makes sense that some classes derived from it and become concrete by overriding all pure virtual functions

- Abstract class specifies interface requirements

- A class derived from an ABC is still abstract if it doesn’t override **ALL** inherited pure virtual functions

Power of Abstract Base Class

```
void draw_shapes(Shape* sarr[], int size)
{
    for(int i=0; i<size; i++)
        sarr[i]->draw();  // draw object circle, triangle, rectangle …..
}
```

draw_shapes : pointer array

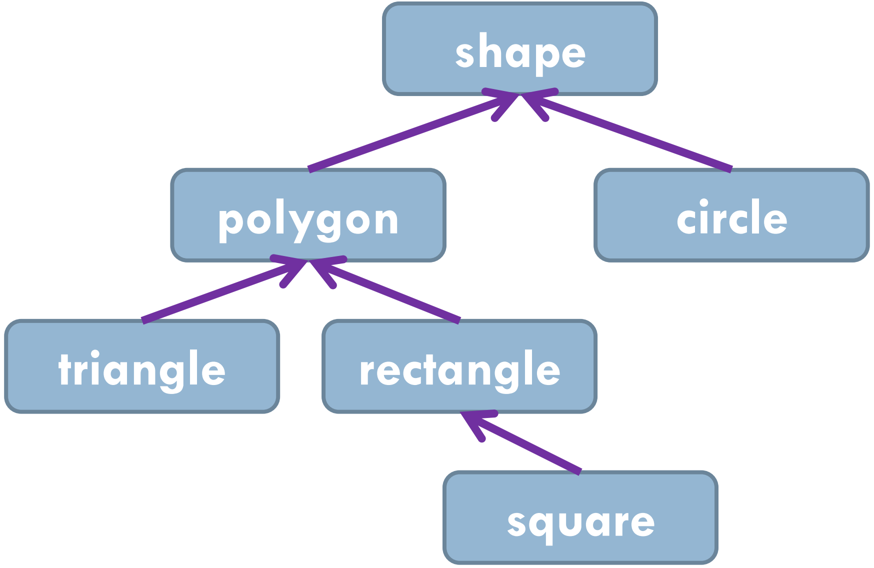
Can correctly call ALL kinds of objects of concrete classes derived from Shape

Exercise(1/2)

- Input: A set of shapes with side-information in a txt file
- Shape type: triangle, rectangle, square and circle
- Output:
  - 1) Calculate the perimeter of each shape.
  - 2) Check if the shapes are legal or illegal, and classify these shapes as the output.
- The rules are the below:
  - For all shapes, all the numbers (length of size and radius) are greater than 0.
  - For triangles, the sum of any two of the sizes is always greater than the other side.
  - For rectangles,
    the up-side = down-side and right-side = left-side
  - For squares, the four side length are identical

Exercise(2/2)

- The inheritance hierarchy should be:
  - shape and polygon is **abstract base class**



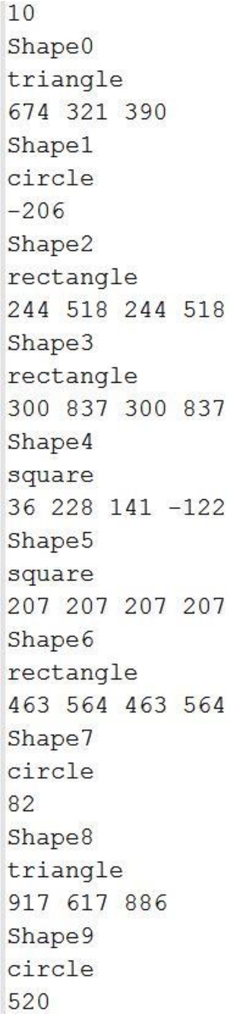
- Set pi = 3.14
- Set the perimeter of the illegal ones **-1**

Format of Input File

```
10 //total number of shapes
Shape0 // [shape name]
triangle
3 3 2 //side lengths
Shape1 //[shape name]
circle
4 //radius
Shape2 //[shape name]
rectangle
3 4 3 4 //always in the order: up, right, down, and left
… …
```

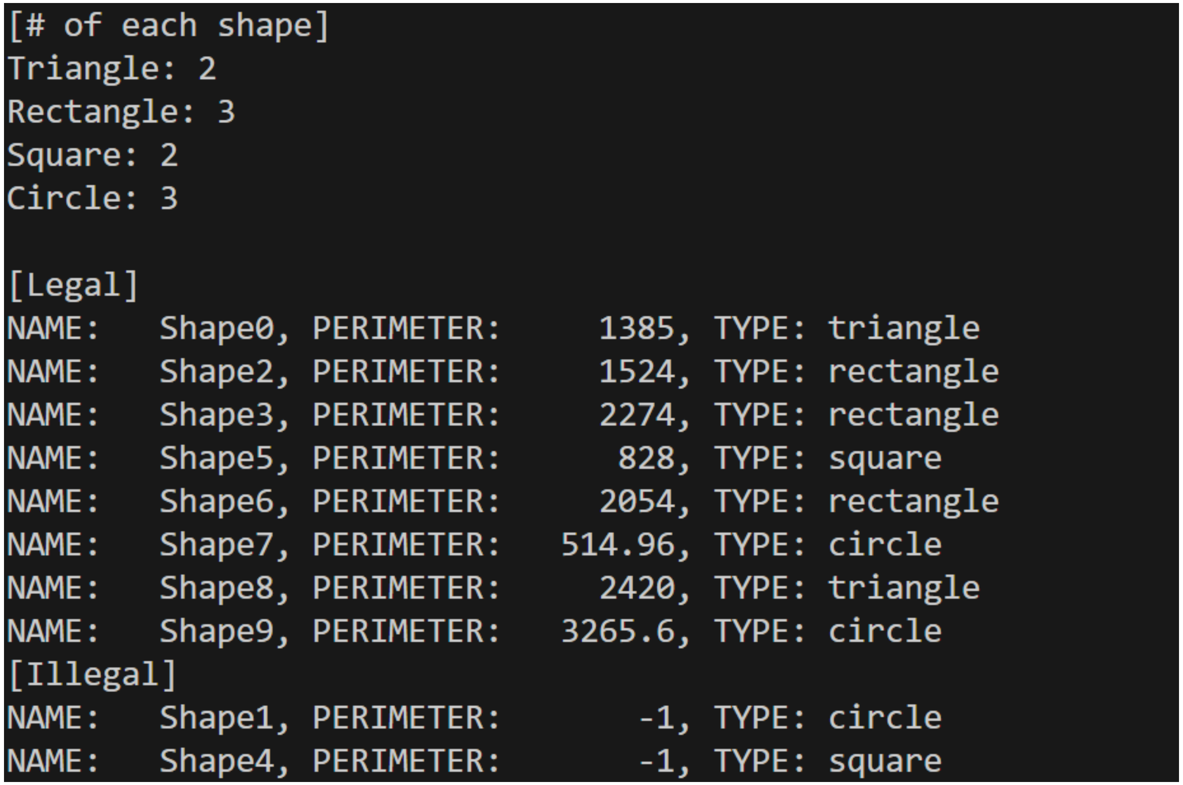
Execution Result (pattern1.txt)

INPUT FILE



```
10
Shape0
triangle
674 321 390
Shape1
circle
-206
Shape2
rectangle
244 518 244 518
Shape3
rectangle
300 837 300 837
Shape4
square
36 228 141 -122
Shape5
square
207 207 207 207
Shape6
rectangle
463 564 463 564
Shape7
circle
82
Shape8
triangle
917 617 886
Shape9
circle
520
```

OUTPUT RESULT



```
[# of each shape]
Triangle: 2
Rectangle: 3
Square: 2
Circle: 3

[Legal]
NAME:    Shape0, PERIMETER:     1385, TYPE: triangle
NAME:    Shape2, PERIMETER:     1524, TYPE: rectangle
NAME:    Shape3, PERIMETER:     2274, TYPE: rectangle
NAME:    Shape5, PERIMETER:      828, TYPE: square
NAME:    Shape6, PERIMETER:     2054, TYPE: rectangle
NAME:    Shape7, PERIMETER:   514.96, TYPE: circle
NAME:    Shape8, PERIMETER:     2420, TYPE: triangle
NAME:    Shape9, PERIMETER:   3265.6, TYPE: circle
[Illegal]
NAME:    Shape1, PERIMETER:       -1, TYPE: circle
NAME:    Shape4, PERIMETER:       -1, TYPE: square
```