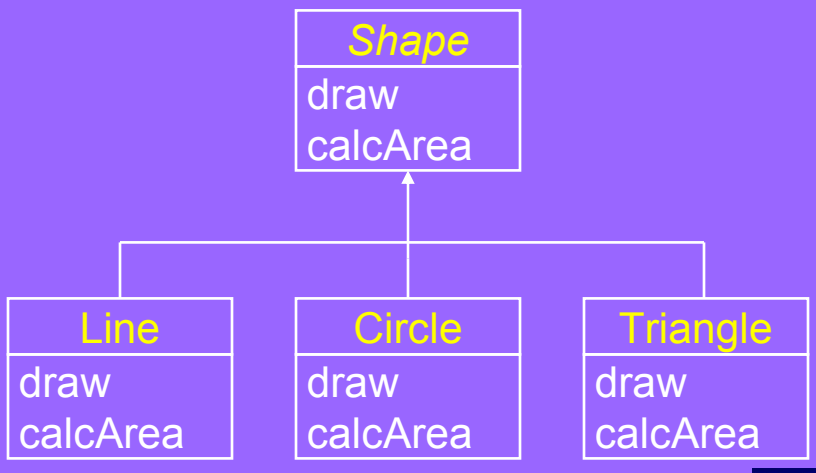
Object-Oriented Programming (OOP)
Lecture No. 28

Problem Statement

- Develop a function that can draw different types of geometric shapes from an array

Shape Hierarchy



Shape Hierarchy

```
class Shape {
  …
protected:
  char _type;
public:
  Shape() { }
  void draw(){ cout << "Shape\n"; }
  int calcArea() { return 0; }
  char getType() { return _type; }
}
```

… Shape Hierarchy

```
class Line : public Shape {
  …
public:
  Line(Point p1, Point p2) {
  …
  }
  void draw(){ cout << "Line\n"; }
}
```

… Shape Hierarchy

```
class Circle : public Shape {
  …
public:
  Circle(Point center, double radius) {
     …
  }
  void draw(){ cout << "Circle\n"; }
  int calcArea() { … }
}
```

… Shape Hierarchy

```
class Triangle : public Shape {
  …
public:
  Triangle(Line l1, Line l2,
                        double angle)
  { … }
  void draw(){ cout << "Triangle\n"; }
  int calcArea() { … }
}
```

Drawing a Scene

```
int main() {
  Shape* _shape[ 10 ];
  Point p1(0, 0), p2(10, 10);
  shape[1] = new Line(p1, p2);
  shape[2] = new Circle(p1, 15);
  …
  void drawShapes( shape, 10 );
  return 0;
}
```

Function drawShapes()

```
void drawShapes(Shape* _shape[],
                int size) {
  for (int i = 0; i < size; i++) {
     _shape[i]->draw();
  }
}
```

Sample Output

```
Shape
Shape
Shape
Shape
…
```

Function drawShapes()

```
void drawShapes(
  Shape* _shape[], int size) {
  for (int i = 0; i < size; i++) {
     // Determine object type with
     // switch & accordingly call
     // draw() method
  }
}
```

Required Switch Logic

```
switch ( _shape[i]->getType() )
{
  case 'L':
     static_cast<Line*>(_shape[i])->draw();
     break;
  case 'C':
     static_cast<Circle*>(_shape[i])
                                ->draw();
     break;
  …
}
```

Equivalent If Logic

```
if ( _shape[i]->getType() == 'L' )
  static_cast<Line*>(_shape[i])->draw();
else if ( _shape[i]->getType() == 'C' )
  static_cast<Circle*>(_shape[i])->draw();
…
```

Sample Output

```
Line
Circle
Triangle
Circle
…
```

Problems with Switch Statement

…Delocalized Code

- Consider a function that prints area of each shape from an input array

Function printArea

```
void printArea(
  Shape* _shape[], int size) {
  for (int i = 0; i < size; i++) {
     // Print shape name.
     // Determine object type with
     // switch & accordingly call
     // calcArea() method.
  }
}
```

Required Switch Logic

```
switch ( _shape[i]->getType() )
{
  case 'L':
     static_cast<Line*>(_shape[i])
              ->calcArea();    break;
  case 'C':
     static_cast<Circle*>(_shape[i])
              ->calcArea();    break;
  …
}
```

...Delocalized Code

- The above switch logic is same as was in function drawArray()
- Further we may need to draw shapes or calculate area at more than one places in code

Other Problems

- Programmer may forget a check
- May forget to test all the possible cases
- Hard to maintain

Solution?

- To avoid switch, we need a mechanism that can select the message target automatically!

Polymorphism Revisited

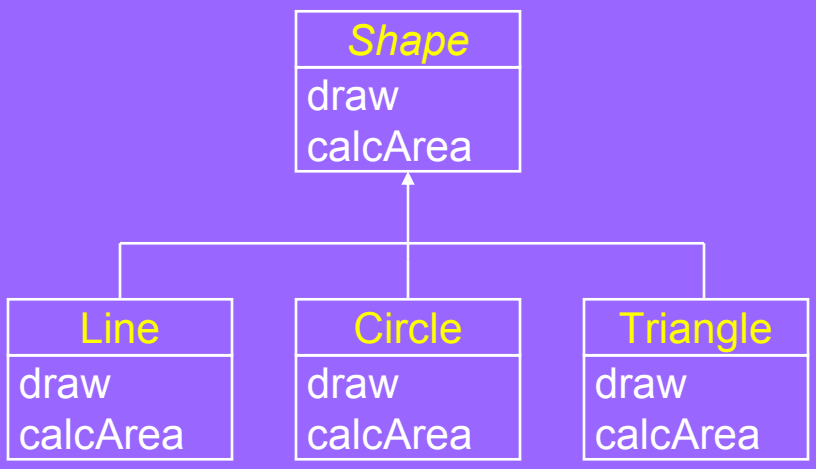
- In OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus)
- Consequently, sender of a message does not need to know the exact class of receiver

Virtual Functions

- Target of a virtual function call is determined at run-time
- In C++, we declare a function virtual by preceding the function header with keyword "virtual"

```
class Shape {
   …
   virtual void draw();
}
```

Shape Hierarchy



...Shape Hierarchy Revisited

No type field

```
class Shape {
  …
  virtual void draw();
  virtual int calcArea();
}
class Line : public Shape {
  …
  virtual void draw();
}
```

... Shape Hierarchy Revisited

```
class Circle : public Shape {
  …
  virtual void draw();
  virtual int calcArea();
}
class Triangle : public Shape {
  …
  virtual void draw();
  virtual int calcArea();
}
```

Function drawShapes()

```
void drawShapes(Shape* _shape[],
                int size) {
    for (int i = 0; i < size; i++) {
        _shape[i]->draw();
  }
}
```

Sample Output

```
Line
Circle
Triangle
Circle
…
```

Function printArea

```
void printArea(Shape* _shape[],
                    int size) {
     for (int i = 0; i < size; i++) {
          // Print shape name
          cout<< _shape[i]
                         ->calcArea();
          cout << endl;
  }
}
```

Static vs Dynamic Binding

- Static binding means that target function for a call is selected at compile time
- Dynamic binding means that target function for a call is selected at run time

Static vs Dynamic Binding

```
Line _line;
_line.draw();     // Always Line::draw
                  // called
Shape* _shape = new Line();
_shape->draw(); // Shape::draw called
          // if draw() is not virtual

Shape* _shape = new Line();
_shape->draw(); // Line::draw called
          // if draw() is virtual
```