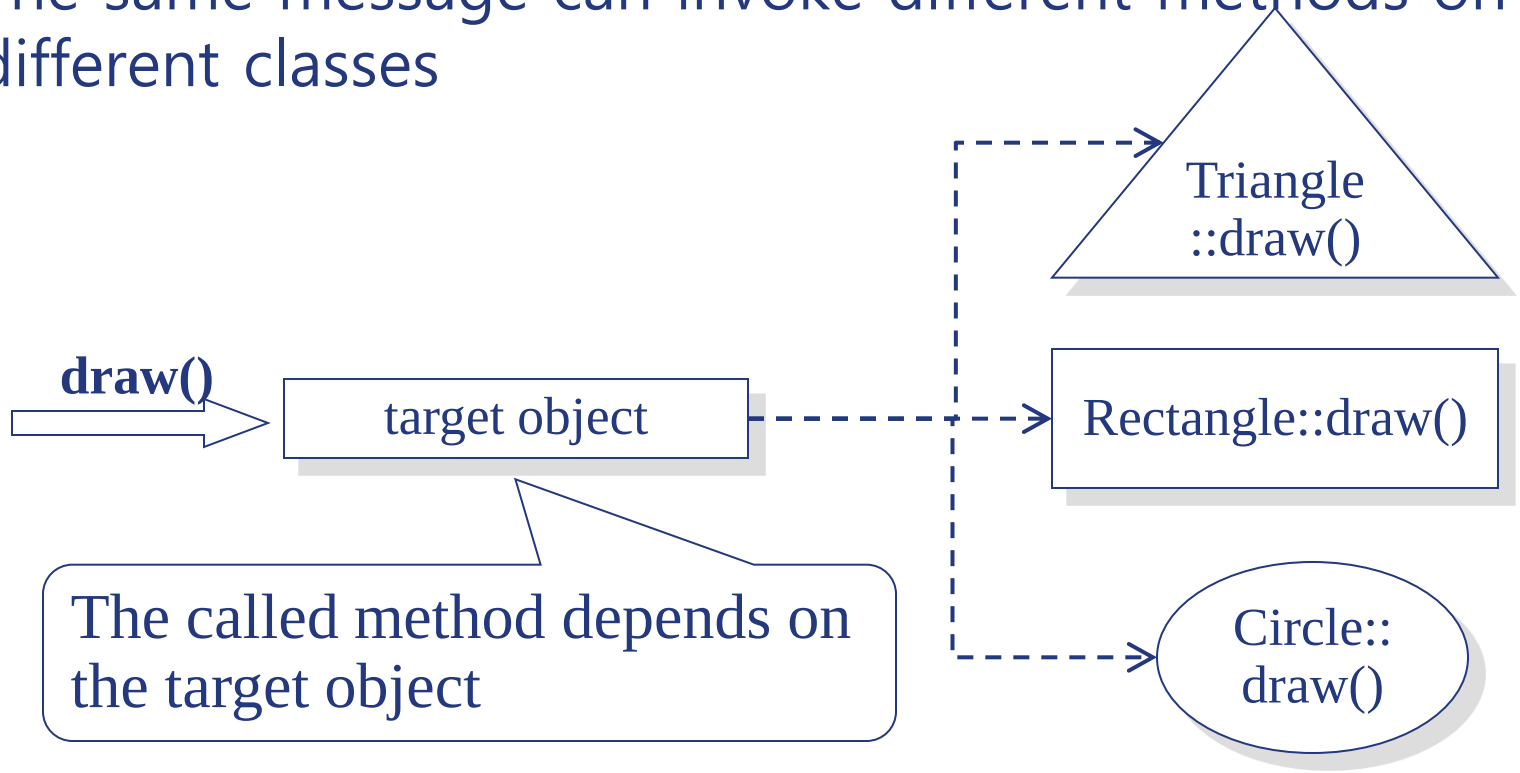
# Topic 11
# Polymorphism

- Polymorphsim– overview
- Virtual function
- Abstract class

# Polymorphism in Programming

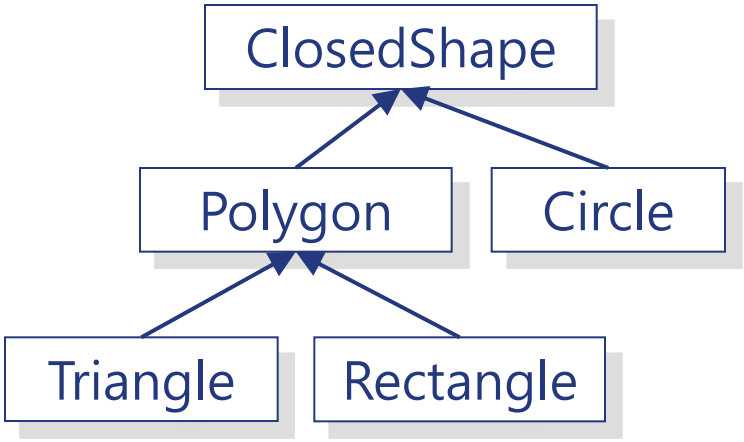
- The same message can invoke different methods on different classes



# Polymorphism in C++

- What object can closedShape point to ?
- What member function is really invoked ?

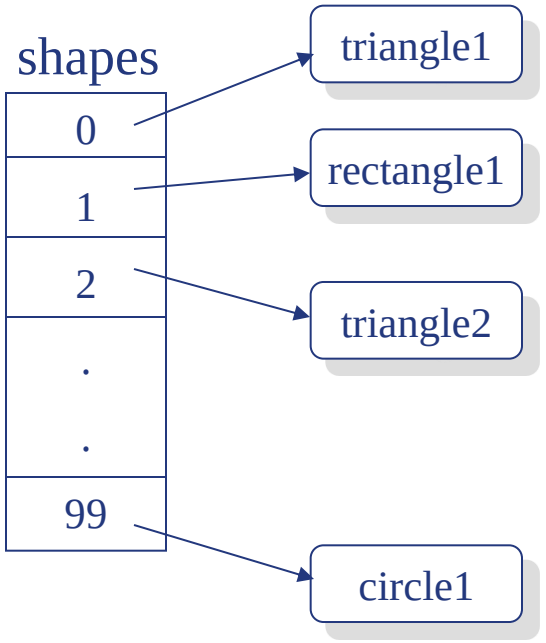
```
void f() {
  ClosedShape* closedShape ;

  closedShape = new Triangle ;
  closedShape->draw() ; // (1)

  closedShape = new Rectangle  ;
  closedShape- >draw() ; // (2)

  closedShape = new Circle ;
  closedShape- >draw() ; // (3)
}
```



# Advantage of Polymorphism

- Consider a function to get the total area of a list of closed shapes.

```
class ClosedShapeList {
  ClosedShape* shapes[100] ;
  // vector<ClosedShape*> shapes(100) ;
  public:
    float getTotalArea() {
      float total_area = 0.0 ;
      for ( int i = 0 ; i < 100 ; i ++ )
        total_area += shapes[i]->getArea() ;
      return total_area ;
  }
} ;
```



# Advantage of Polymorphism

- The function will be like this without polymorphism

```
float ClosedShapeList::getTotalArea() {
      float total_area = 0.0 ;
      for ( int i = 0 ; i < 100 ; i ++ ) {
        switch ( shapes[i]->type ) {
          case TRIANGLE :
             Triangle *t = static_cast<Triangle*> shapes[i];
             area = t->getArea() ;
             break;
          case RECTANGLE :       …
          case CIRCLE :          …
        }
        total_area += area ;
      }
      return total_area ;
} ;
```

# Advantage of Polymorphism

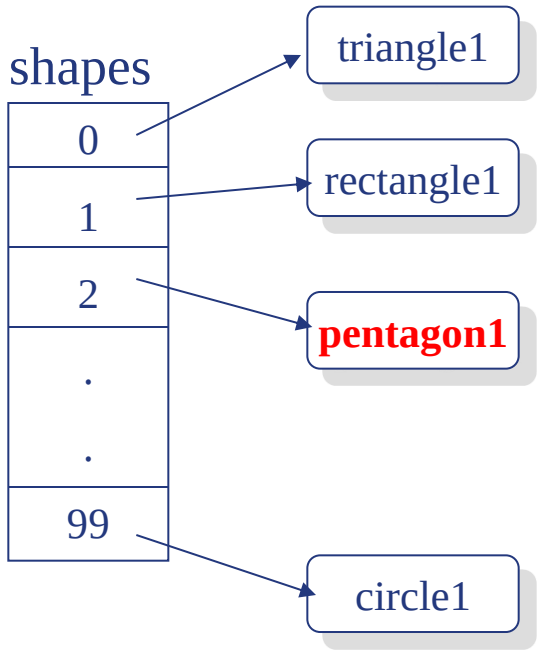
- If you want to consider new Shape, for example Pentagon
  - define Pentagon as a derived class of Polygon
  - override an operation calcArea in Polygon

```
class Pentagon: public Polygon {
  public:
    float getArea() {
      .....
    }
} ;
```

# Advantage of Polymorphism

- no change to ClosedShapeList::getTotalArea()

```
class ClosedShapeList {
  ClosedShape* shapes[100] ;
  // vector<ClosedShape*> shapes(100) ;
public:
    float getTotalArea() {
      float total_area = 0.0 ;
      for ( int i = 0 ; i < 100 ; i ++ )
        total_area += shapes[i]->getArea() ;
      return total_area ;
    }
} ;
```



# Advantage of Polymorphism

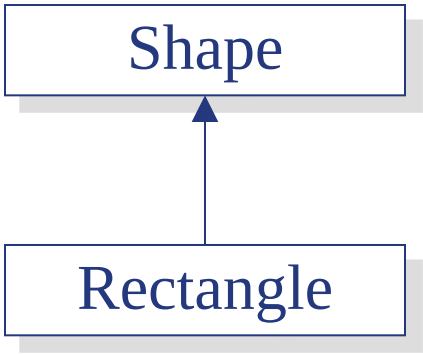
- Without polymorphism, ClosedShapeList::getTotalArea() should be changed

```
float ClosedShapeList::getTotalArea() {
        float total_area = 0.0 ;
        for ( int i = 0 ; i < 100 ; i ++ ) {
          switch ( shapes[i]->type ) {
            // cases of TRIANGLE, RECTANGLE, CIRCLE…
           case PENTAGON:
               Pentagon *p = static_cast<Pentagon*> shapes[i];
               area = p->getArea() ;
               break;
          }
           total_area += area ;
        }
        return total_area ;
} ;
```

# Substitutability Principle

- An object of a subclass is also an object of the superclass

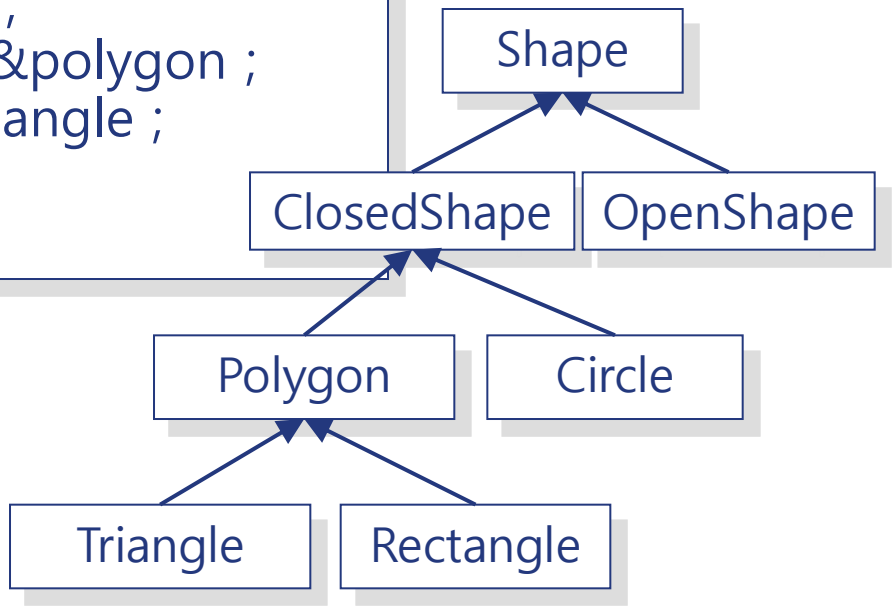
```
void f() {
  Rectangle r ;
  Shape s ;
  Shape* pShape ;

  pShape = &s ;

  pShape = &r ;
}
```




pShape can point to any descendent of Shape in addition to Shape

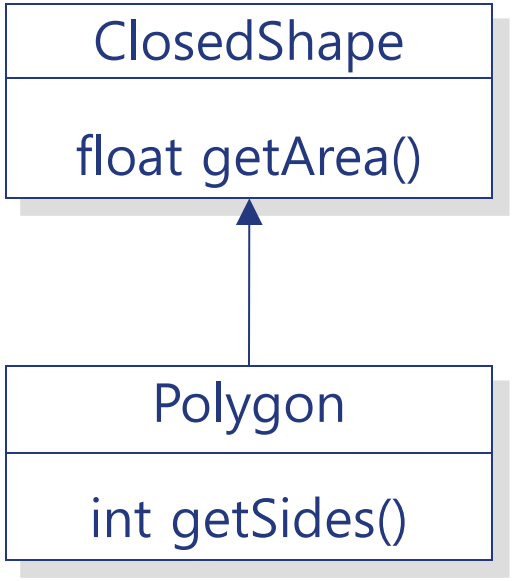
# Substitutability Principle: Example

```
void f() {
  ClosedShape closed_figure ;
  Polygon  polygon ;
  Triangle triangle ;
  Circle circle ;

  Shape* pShape = &closedShape ;
  ClosedShape*  pClosedShape = &polygon ;
  OpenShape* pOpenShape = &triangle ;
  Polygon* pPolygon = &circle ;
}
```



# Substitutability Principle

- However, only members of the superclass can be invoked.




```
Base* pBase = new Derived ;

pBase->base-members()
```

```
void f() {
  ClosedShape*  pClosedShape = new Polygon ;

   pClosedShape->getArea() ; // OK
   pClosedShape->getSides() ; // ERROR
   // getSides() not declared in ClosedShape
}
```

# Virtual Function in C++

- Basically member function is statically bound.
- So, substitutability principle does not applied.

```
class Polygon {
  public:
    float getArea()  ;
} ;
```

```
class Triangle: public Polygon {
  public:
    float getArea()  ;
} ;
```

*polygon* can point to any object of Polygon and its derived classes

```
void f()
{
  Polygon* polygon ;
  polygon = new Polygon ;
  polygon->getArea() ;
  // Polygon::getArea()

  polygon = new Triangle ;
  polygon->getArea() ;
  // Polygon::getArea(), not Triangle::getArea()
}
```

# Virtual Function in C++

- Only member function with **virtual** is dynamically bound

```
class Polygon {
  public:
    virtual float getArea()  ;
} ;
```

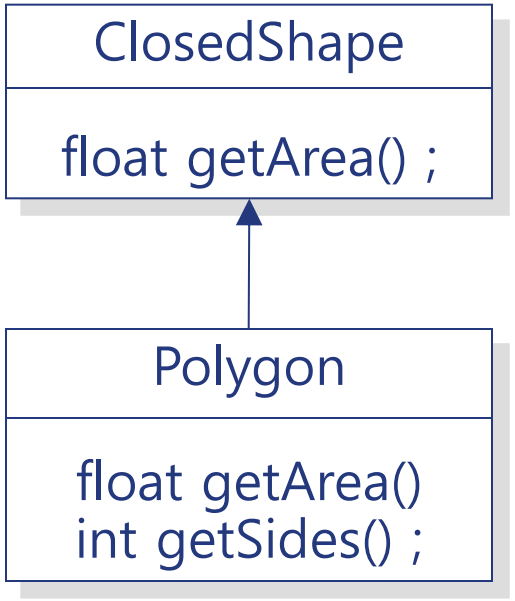
```
class Triangle: public Polygon {
  public:
    virtual float getArea()  ;
} ;
```

```
void f() {
  Polygon* polygon ;
  polygon = new Polygon ;
  polygon->getArea() ;
  // Polygon::getArea()

  polygon = new Triangle ;
  polygon->getArea() ;
  // Triangle::getArea()
}
```

# IS-A Relation

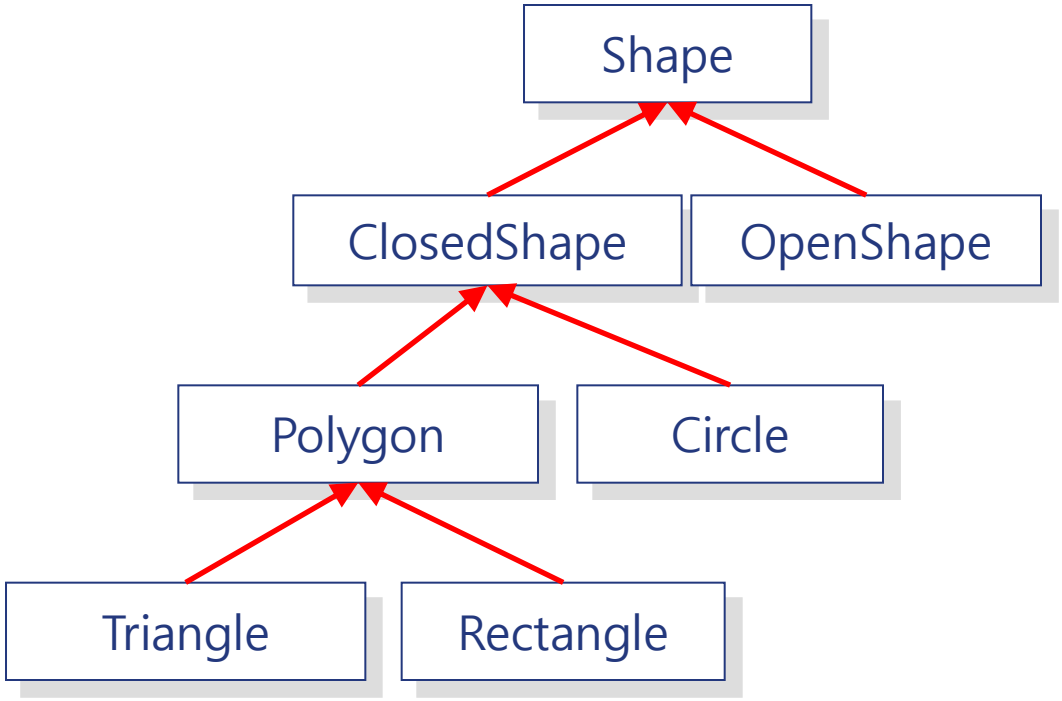
- Inheritance should represent "**is-a**" relation;
- In other words,
  - **a subclass is a superclass**, or
  - **a subclass is a kind of a superclass**.

- A Polygon "IS-A" a ClosedShape



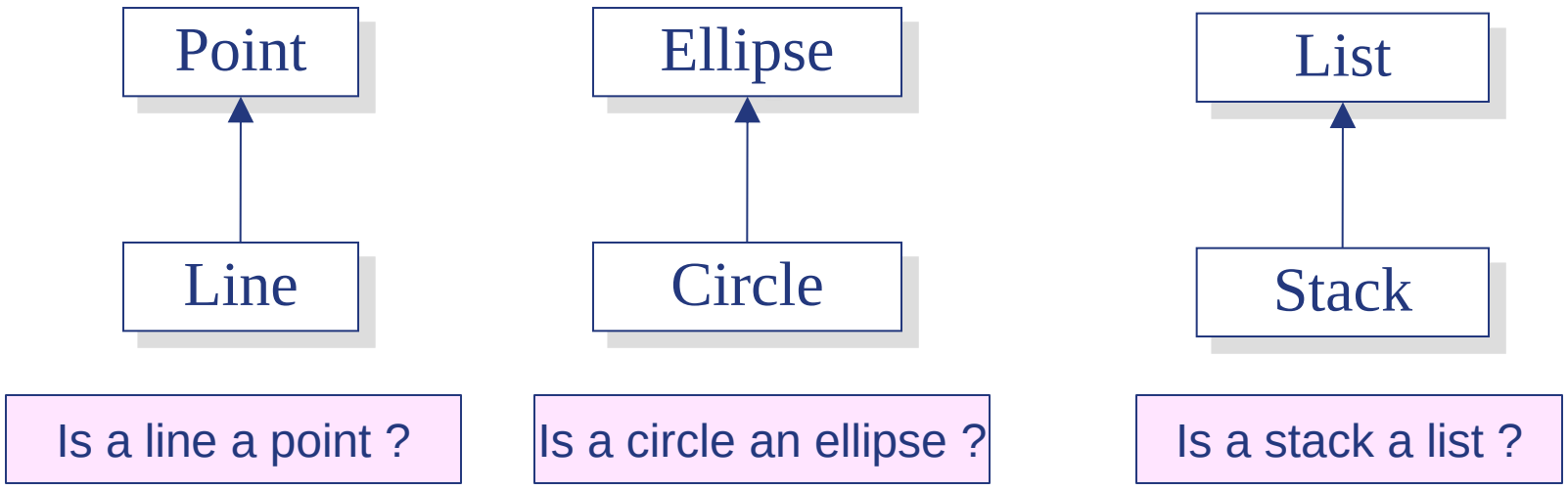
# IS-A Relation: An Example

- Check whether each inheritance relationship is properly used.



# Improper Inheritance: Examples

- If an inheritance does not conform to the is-a relation, it is not proper, which may cause serious problems.
- Are the inheritances below properly used ?
  - Determine them on the basis of IS-A concept.
  - If not, give spefic reasons behind such decision.



# Virtual Function in C++

- **Overriding** virtual function should have the same signature as the overridden one.
- The signature includes constness.

```
class Person {
  private:
    string name;
    int age ;
    string address ;
  public:
    virtual void print() const ;
} ;
```

```
class Student: public Person {
  private:
    ...
  public:
    virtual void print(int) ;
    virtual void print() ;
} ;
```

Student::print(int) and non-const print() are not overriding Person::print(), but new virtual functions

# Virtual Function in C++

- Overriding without virtual can lead to an unintended program behavior

```
class ClosedShape{
  public:
    void draw() ;
} ;
class Triangle : public ClosedShape {
  public:
    void draw() ;
} ;
…
class Circle : public ClosedShape {
  public:
    void draw() ;
} ;
```

only ClosedFigure::draw() is invoked three times

```
void f() {
  ClosedShape* closedShape ;

  closedShape = new Triangle ;
  closedShape->draw() ;

  closedShape = new Rectangle  ;
  closedShape- >draw() ;

  closedShape = new Circle ;
  closedShape- >draw() ;
}
```

# Why Use Overriding ?

1) Adaptation: to correctly implement the behavior of subclass

```
class Employee{
  protected:
    float rate ;
    in workDays ;
  public:
    virtual float getBonus() const { return rate * workDays ; }
} ;
```

```
class Manager: public Employee{
    vector< Employee *> group;
  public:
    float getBonus() const { return rate*work_days + rate * 0.1 * group.size() ; }
} ;
```

# Why Use Overriding ?

2) Optimization: to improve the performance

```
class Polygon {
  protected:
    vector<Point> points ;
  public:
    virtual float getArea() const { /* general algorithm for polygon */ }
} ;
```



```
class Triangle: public Polygon {
  public:
    float getArea() const { /* faster algorithm specific for Triangle  */ }
} ;
```

# Overriding: Requirements

- Overriding operation should conform to the original behavior of the overridden operation

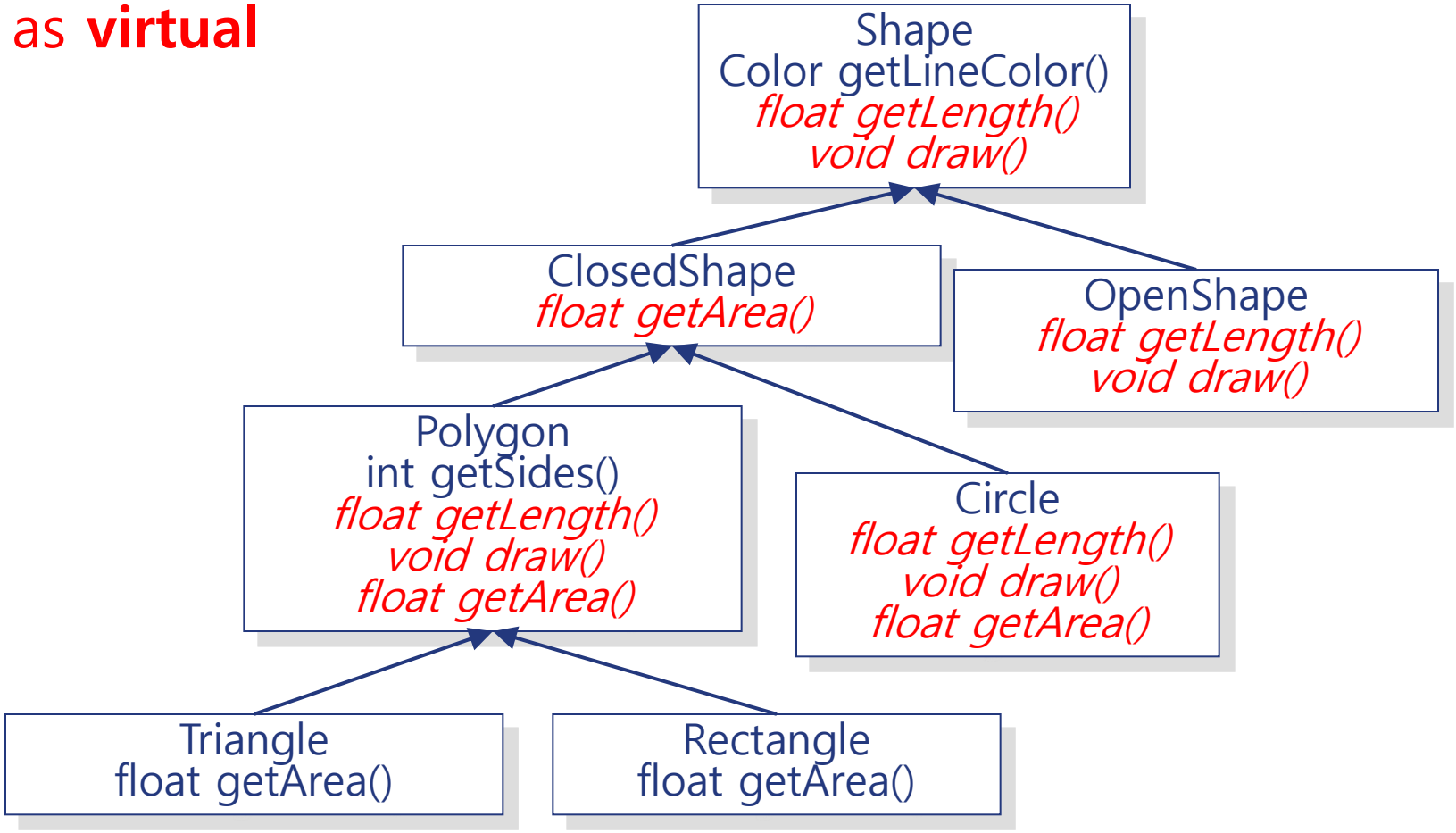
```
class Manager: public Employee{
    vector< Employee *> group;
  public:
    float getBonus() const { return regular salary }
} ;
```

```
void f() {
  Employee man ;
  float bonus1 = man.getBonus() ;
  Manager manager ;
  float bonus2 = manager.getBonus() ;
}
```

confusing; show something different from our expectation

# Inheritance Hierarchy for Shape

- In essence, overridden operations should be declared as **virtual**



# Abstract Class

- Some classes, such as Shape, represent abstract concepts for which objects cannot exist.

```
void f () {
  Shape shape ; // awkward : we don't know what it is
}
```

- In addition, some member functions cannot be implemented.

```
class Shape{
    Color lineColor ;
  public:
    Color getLineColor() const { return lineColor ; }
    virtual float getLength()  const { cerr << "Shape::getLength()" ; }
    virtual void draw() { cerr << "Shape::draw()" ; }
} ;
```

We can neither get length nor draw without knowing its shape

# Pure Virtual Function

- **Pure virtual function** is a virtual function with an initializer **=0**

```
class Shape {
    Color lineColor ;
  public:
    Color getLineColor() const { return lineColor ; }
    virtual float getLength()  const = 0 ;
    virtual void draw() = 0 ;
} ;
```

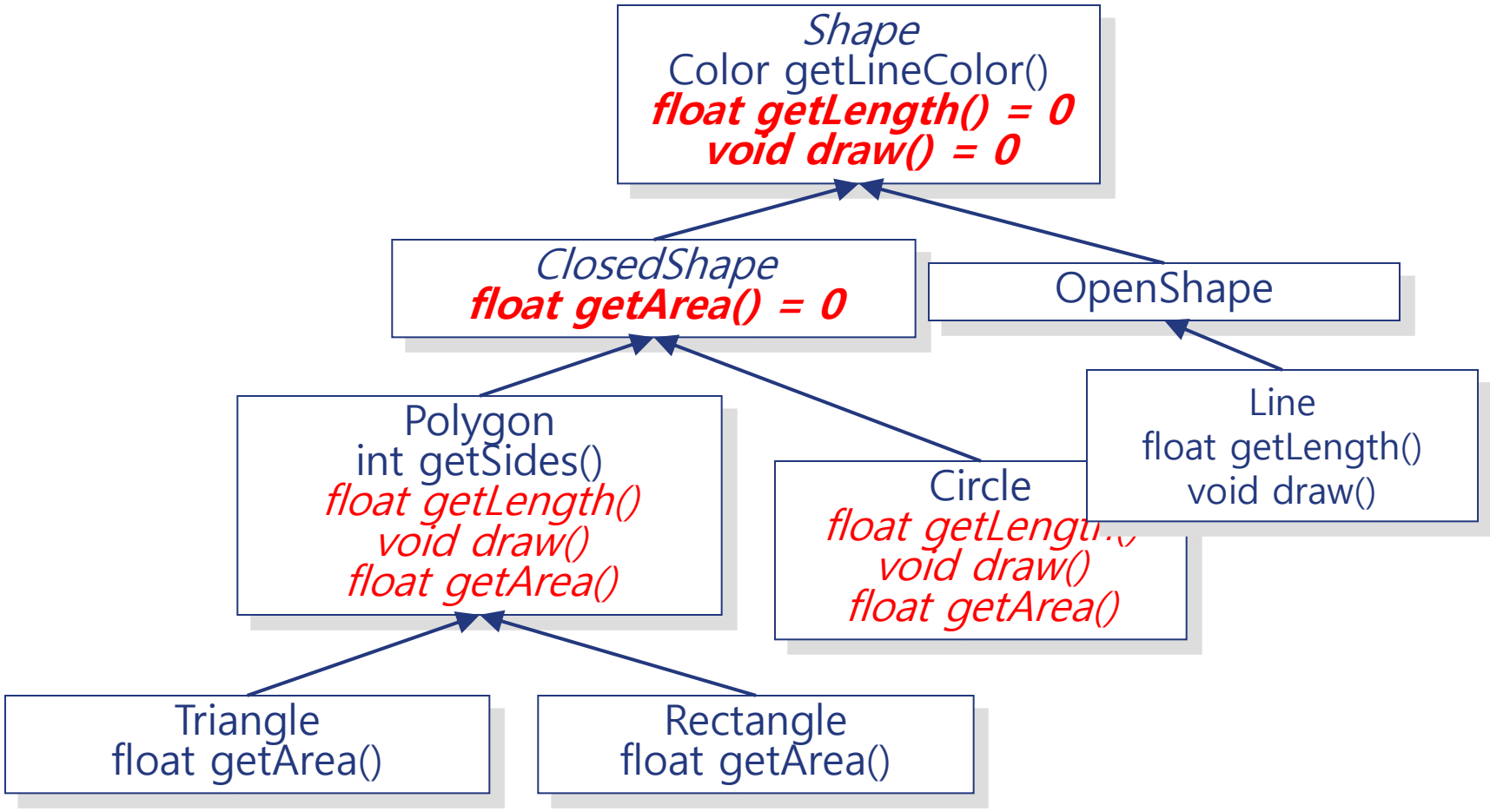
```
class ClosedShape : public Shape {
    Color fillColor ;
  public:
    Color getFillColor() const { return fillColor ; }
    virtual float getArea()  const = 0 ;
} ;
```

# Abstract Class

- Abstract Class: class with one or more pure virtual functions.
- It is not allowed to create an object from an abstract class.

```
void f() {
  Shape shape ; // ERROR:
  ClosedShape* pClosedShape = new ClosedShape ; // ERROR
}
```

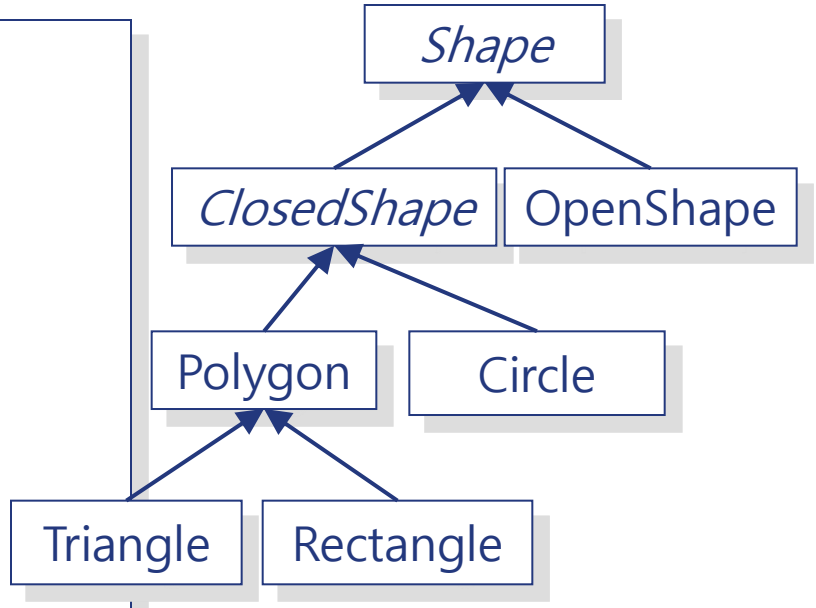
# Inheritance Hierarchy for Shape: Revised with Abstract Class



# Abstract Class

- However, a pointer or reference of an abstract class is possible.

```
void draw(Shape* pShape) {
    pShape->draw() ;
}
float getLength(const Shape& rShape) {
  return rShape.getLength() ;
}
int main() {
  Shape* pShape = new Polygon(5) ;
  draw(pShape) ;
  ClosedShape* pClosedShape = new Triangle ;
  draw(pClosedShape) ;
  Rectangle r ;
  cout << getLength(r) ;
}
```



# Abstract Class in C++

- A pure virtual function that is not defined in a derived class remains a pure virtual function.
- In that case, the derived class is also an abstract class

```
class X {
  public:
    virtual void f() = 0 ;
    virtual void g() = 0 ;
} ;

X a ;  // error
// X::f(), X::g() are pure
```

```
class Y : public X {
  public:
    void f() ;
    // override X::f()
} ;

Y b ; // error
// X::g() still pure
```

```
class Z : public Y {
  public:
    void g() ;
    // override X::g()
} ;

Z c ; // ok
// Z is not abstract any more
```

# Abstract Class for Interface

- Abstract classes are used to define an interface without any implementation details
- e. g.) class CharacterDevice provides an common interface independent of actual devices.

```
class CharacterDevice {
  public:
    virtual int open() = 0 ;
    virtual int close(const char*) = 0 ;
    virtual int read(const char*, int) = 0 ;
    virtual int write(const char*, int) = 0 ;
} ;
```

- Such an abstract class is similar to **interface** in Java.

# Generic Container Class

- Implement ShapeList for holding various Shapes

```
int main() {
   Point p1(0, 0), p2(0, 10), p3(20, 20), p4(20, 30) ;

   Shape* r = new Rectangle(p1, p2, p3, p4) ;
   Shape* t = new Triangle(p1, p2, p3) ;

   ShapeList list ;
   list.addShape(r) ;
   list.addShape(t) ;

   list.print() ;
   cout << list.getTotalArea() << endl ;
}
```

```
Rectangle: (0, 0)(0, 10)(20, 20)(20, 30)
Triangle: (0, 0)(0, 10)(20, 20)
XXXX
```

# Assumptions

```
enum Color {RED, BLUE, YELLOW} ;


class Shape {
    Color lineColor ;
public:
   Color getLineColor() const { return lineColor ; }
    virtual Shape* clone() const = 0 ;
    virtual void print() const = 0 ;
    virtual float getLength() const = 0 ;
} ;
```