CS100 Lecture 22

Inheritance and Polymorphism II

Contents

- Abstract base class
- The "is-a" relationship revisited (*Effective C++* Item 32)
- Inheritance of interface vs inheritance of implementation (*Effective C++* Item 34)

Abstract base class

Shapes

Define different shapes: Rectangle, Triangle, Circle, ...

Suppose we want to draw things like this:

```
void drawThings(ScreenHandle &screen,
                const std::vector<std::shared_ptr<Shape>> &shapes) {
  for (const auto &shape : shapes)
    shape->draw(screen);
}
```

and print information:

```
void printShapeInfo(const Shape &shape) {
  std::cout << "Area: " << shape.area()
            << "Perimeter: " << shape.perimeter() << std::endl;
}
```

Shapes

Define a base class `Shape` and let other shapes inherit it.

```
class Shape {
public:
  Shape() = default;
  virtual void draw(ScreenHandle &screen) const;
  virtual double area() const;
  virtual double perimeter() const;
  virtual ~Shape() = default;
};
```

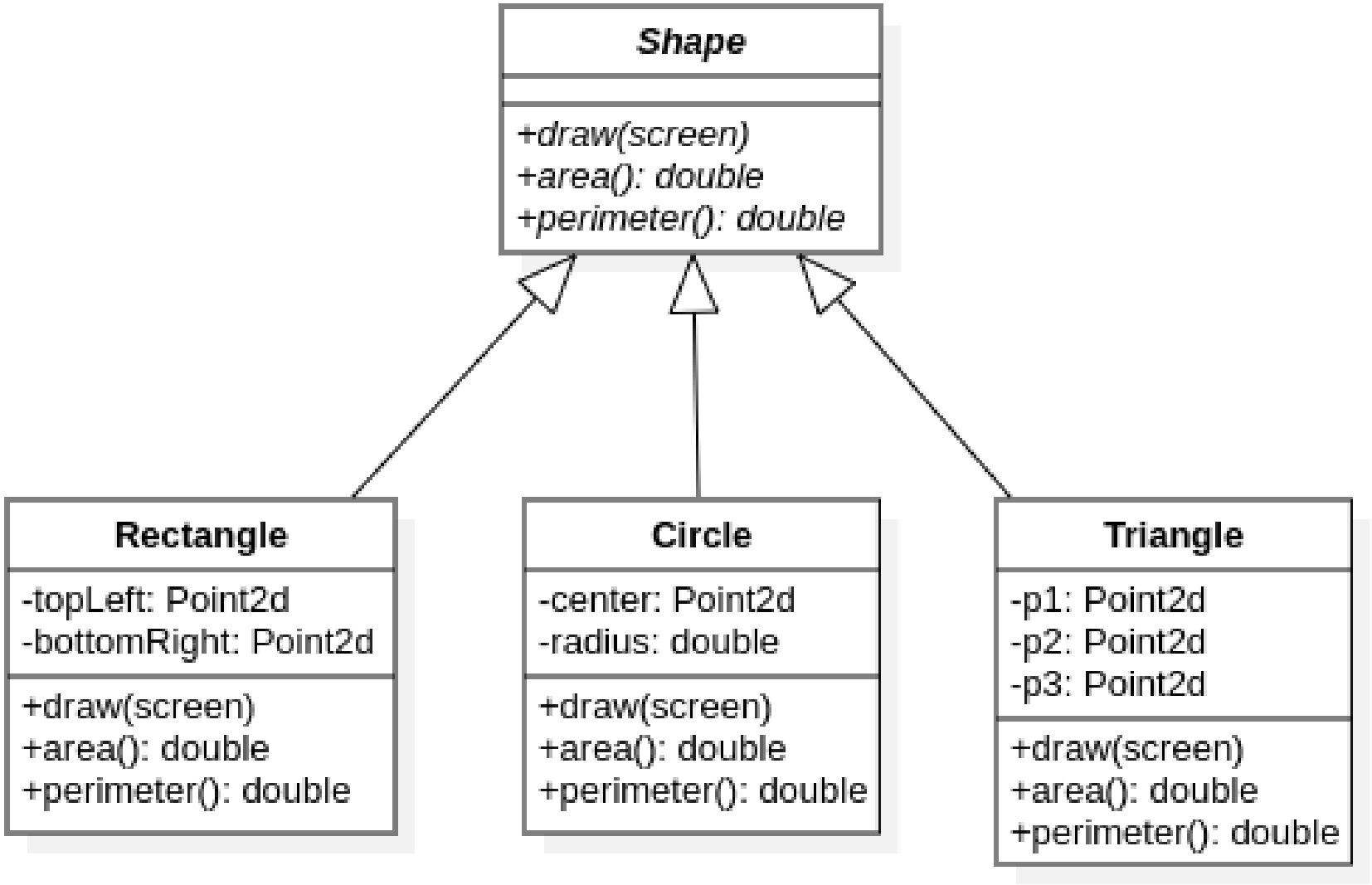
Different shapes should define their own `draw`, `area` and `perimeter`, so these functions should be `virtual`.

Shapes

```
class Rectangle : public Shape {
  Point2d mTopLeft, mBottomRight;

public:
  Rectangle(const Point2d &tl, const Point2d &br)
      : mTopLeft(tl), mBottomRight(br) {} // Base is default-initialized
  void draw(ScreenHandle &screen) const override { /* ... */ }
  double area() const override {
    return (mBottomRight.x - mTopLeft.x) * (mBottomRight.y - mTopLeft.y);
  }
  double perimeter() const override {
    return 2 * (mBottomRight.x - mTopLeft.x + mBottomRight.y - mTopLeft.y);
  }
};
```

Shapes


Shape
+draw(screen)
+area(): double
+perimeter(): double
Rectangle
-topLeft: Point2d
-bottomRight: Point2d
+draw(screen)
+area(): double
+perimeter(): double
Circle
-center: Point2d
-radius: double
+draw(screen)
+area(): double
+perimeter(): double
Triangle
-p1: Point2d
-p2: Point2d
-p3: Point2d
+draw(screen)
+area(): double
+perimeter(): double

Pure `virtual` functions

How should we define `Shape::draw`, `Shape::area` and `Shape::perimeter`?

- For the general concept "Shape", there is no way to determine the behaviors of these functions.

Pure `virtual` functions

How should we define `Shape::draw`, `Shape::area` and `Shape::perimeter`?

- For the general concept "Shape", there is no way to determine the behaviors of these functions.
- Direct call to `Shape::draw`, `Shape::area` and `Shape::perimeter` should be forbidden.
- We shouldn't even allow an object of type `Shape` to be instantiated! The class `Shape` is only used to **define the concept "Shape" and required interfaces**.

Pure `virtual` functions

If a `virtual` function does not have a reasonable definition in the base class, it should be declared as **pure** `virtual` by writing `=0`.

```
class Shape {
public:
  virtual void draw(ScreenHandle &) const = 0;
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual ~Shape() = default;
};
```

Any class that has a **pure** `virtual` **function** is an **abstract class**. Pure `virtual` functions (usually) cannot be called, and abstract classes cannot be instantiated.

Pure `virtual` functions and abstract classes

Any class that has a **pure `virtual` function** is an **abstract class**. Pure `virtual` functions (usually) cannot be called, and abstract classes cannot be instantiated.

```
Shape shape; // Error.
Shape *p = new Shape; // Error.
auto sp = std::make_shared<Shape>(); // Error.
std::shared_ptr<Shape> sp2 = std::make_shared<Rectangle>(p1, p2); // OK.
```

We can define pointer or reference to an abstract class, but never an object of that type!

Pure `virtual` functions and abstract classes

A non-pure `virtual` function **must be defined**. Otherwise, the compiler will fail to generate necessary runtime information (the virtual table), which leads to an error.

```
class X {
  virtual void foo(); // Declaration, without a definition
  // Even if `foo` is not used, this will lead to an error.
};
```

Linkage error:

```
/usr/bin/ld: /tmp/ccV9TNfM.o: in function `main':
a.cpp:(.text+0x1e): undefined reference to `vtable for X'
```

Make the interface robust, not error-prone.

Is this good?

```
class Shape {
public:
  virtual double area() const {
    return 0;
  }
};
```

What about this?

```
class Shape {
public:
  virtual double area() const {
    throw std::logic_error{"area() called on Shape!"};
  }
};
```

Make the interface robust, not error-prone.

```
class Shape {
public:
  virtual double area() const {
    return 0;
  }
};
```

If `Shape::area` is called accidentally, the error will happen ***silently***!

Make the interface robust, not error-prone.

```
class Shape {
public:
  virtual double area() const {
    throw std::logic_error{"area() called on Shape!"};
  }
};
```

If `Shape::area` is called accidentally, an exception will be raised.

However, **a good design should make errors fail to compile**.

[Best practice] If an error can be caught in compile-time, don't leave it until run-time.

Polymorphism (多态)

Polymorphism: The provision of a single interface to entities of different types, or the use of a single symbol to represent multiple different types.

- Run-time polymorphism: Achieved via **dynamic binding**.
- Compile-time polymorphism: Achieved via **function overloading**, **templates**, **concepts (since C++20)**, etc.

Run-time polymorphism:

```
struct Shape {
  virtual void draw() const = 0;
};
void drawStuff(const Shape &s) {
  s.draw();
}
```

Compile-time polymorphism:

```
template <typename T>
concept Shape = requires(const T x) {
  x.draw();
};
void drawStuff(Shape const auto &s) {
  s.draw();
}
```