

CS100 Lecture 22

Inheritance and Polymorphism II

Contents

- Abstract base class
- More on the "is-a" relationship (*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;  
};
```

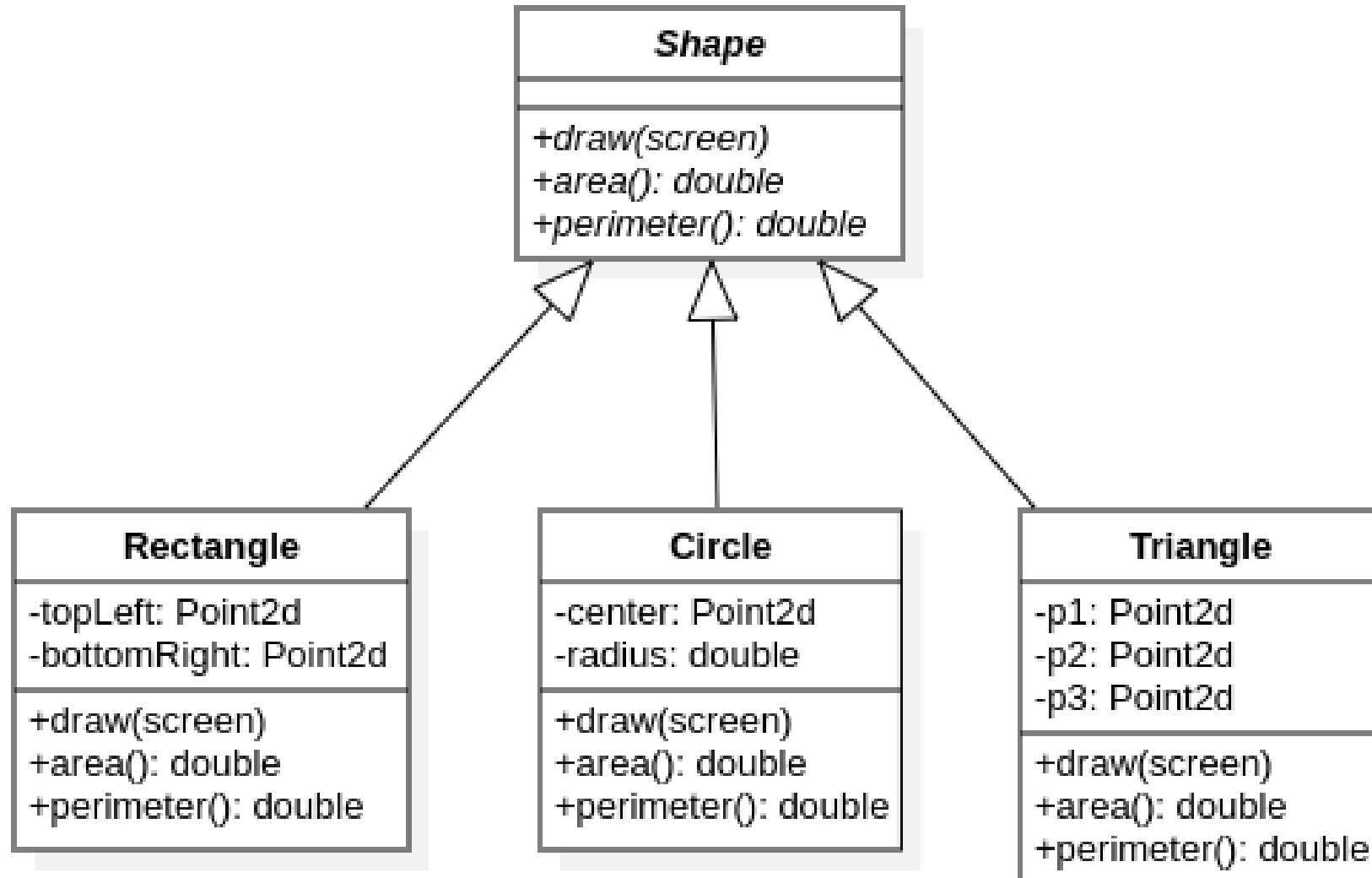
Different shapes should define their own `draw`, `area` and `perimeter`, so these functions should be `virtual`.

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



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 ¹, 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 ¹, 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

An impure `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();  
}
```


More on the "is-a" relationship

Effective C++ Item 32

Public inheritance: The "is-a" relationship

By writing that class `D` publicly inherits from class `B`, you are telling the compiler (as well as human readers of your code) that

- Every object of type `D` *is* also *an* object of type `B`, but not vice versa.
- `B` represents a **more general concept** than `D`, and that `D` represents a **more specialized concept** than `B`.

More specifically, you are asserting that **anywhere an object of type `B` can be used, an object of type `D` can be used just as well.**

- On the other hand, if you need an object of type `D`, an object of type `B` won't do.

Example: Every student *is a* person.

```
class Person { /* ... */ };  
class Student : public Person { /* ... */ };
```

- Every student *is a* person, but not every person is a student.
- Anything that is true of a person is also true of a student:
 - A person has a date of birth, so does a student.
- Something is true of a student, but not true of people in general.
 - A student is enrolled in a particular school, but a person may not.

The notion of a person is **more general** than is that of a student; a student is a **specialized type** of person.

Example: Every student *is a* person.

The **is-a** relationship: Anywhere an object of type `Person` can be used, an object of type `Student` can be used just as well, **but not vice versa**.

```
void eat(const Person &p);    // Anyone can eat.
void study(const Student &s); // Only students study.
Person p;
Student s;
eat(p);    // Fine. `p` is a person.
eat(s);    // Fine. `s` is a student, and a student is a person.
study(s);  // Fine.
study(p);  // Error! `p` isn't a student.
```

Your intuition can mislead you.

- A penguin is a bird.
- A bird can fly.

If we naively try to express this in C++, our effort yields:

```
class Bird {  
public:  
    virtual void fly();           // Birds can fly.  
    // ...  
};  
class Penguin : public Bird { // A penguin is a bird.  
    // ...  
};
```

```
Penguin p;  
p.fly();    // Oh no!! Penguins cannot fly, but this code compiles!
```

No. Not every bird can fly.

In general, birds have the ability to fly.

- Strictly speaking, there are several types of non-flying birds.

Maybe the following hierarchy models the reality much better?

```
class Bird { /* ... */ };
class FlyingBird : public Bird {
    virtual void fly();
};
class Penguin : public Bird {    // Not FlyingBird
    // ...
};
```

No. Not every bird can fly.

Maybe the following hierarchy models the reality much better?

```
class Bird { /* ... */ };  
class FlyingBird : public Bird {  
    virtual void fly();  
};  
class Penguin : public Bird {    // Not FlyingBird  
    // ...  
};
```

- **Not necessarily.** If your application has much to do with beaks and wings, and nothing to do with flying, the original two-class hierarchy might be satisfactory.
- **There is no one ideal design for every software.** The best design depends on what the system is expected to do.

What about report a runtime error?

```
void report_error(const std::string &msg); // defined elsewhere
class Penguin : public Bird {
public:
    virtual void fly() {
        report_error("Attempt to make a penguin fly!");
    }
};
```


What about report a runtime error?

```
void report_error(const std::string &msg); // defined elsewhere
class Penguin : public Bird {
public:
    virtual void fly() { report_error("Attempt to make a penguin fly!"); }
};
```

No. This does not say "Penguins can't fly." This says **"Penguins can fly, but it is an error for them to actually try to do it."**

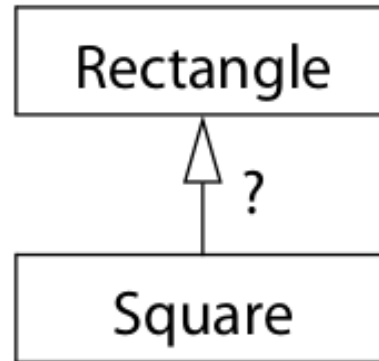
To actually express the constraint "Penguins can't fly", you should prevent the attempt from **compiling**.

```
Penguin p;
p.fly(); // This should not compile.
```

[Best practice] Good interfaces prevent invalid code from compiling.

Example: A square *is a* rectangle.

Should class `Square` publicly inherit from class `Rectangle` ?



Example: A square *is a* rectangle.

Consider this code.

```
class Rectangle {  
public:  
    virtual void setHeight(int newHeight);  
    virtual void setWidth(int newWidth);  
    virtual int getHeight() const;  
    virtual int getWidth() const;  
    // ...  
};  
void makeBigger(Rectangle &r) {  
    r.setWidth(r.getWidth() + 10);  
}
```

```
class Square : public Rectangle {  
    // A square is a rectangle,  
    // where height == width.  
    // ...  
};  
  
Square s(10); // A 10x10 square.  
makeBigger(s); // Oh no!
```

Is this really an "is-a" relationship?

We said before that the "is-a" relationship means that **anywhere an object of type B can be used, an object of type D can be used just as well.**

However, something applicable to a rectangle is not applicable to a square!

Conclusion: Public inheritance means "is-a". Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object.

Inheritance of interface vs inheritance of implementation

Effective C++ Item 34

Example: Airplanes for XYZ Airlines.

Suppose XYZ has only two kinds of planes: the Model A and the Model B, and both are flown in exactly the same way.

```
class Airplane {  
public:  
    virtual void fly(const Airport &destination) {  
        // Default code for flying an airplane to the given destination.  
    }  
};  
class ModelA : public Airplane { /* ... */ };  
class ModelB : public Airplane { /* ... */ };
```

- `Airplane::fly` is declared `virtual` because *in principle*, different airplanes should be flown in different ways.
- `Airplane::fly` is defined, to avoid copy-and-pasting code in the `ModelA` and `ModelB` classes.

Example: Airplanes for XYZ Airlines.

Now suppose that XYZ decides to acquire a new type of airplane, the Model C, **which is flown differently from the Model A and the Model B.**

XYZ's programmers add the class `ModelC` to the hierarchy, but forget to redefine the `fly` function!

```
class ModelC : public Airplane {  
    // `fly` is not overridden.  
    // ...  
};
```

This surely leads to a disaster:

```
auto pc = std::make_unique<ModelC>();  
pc->fly(PVG); // No! Attempts to fly Model C in the Model A/B way!
```

Impure virtual function: Interface + default implementation

The problem here is not that `Airplane::fly` has default behavior, but that `ModelC` was allowed to inherit that behavior **without explicitly saying that it wanted to**.

*** By defining an impure virtual function, we have the derived class inherit a function *interface as well as a default implementation*.**

- Interface: Every class inheriting from `Airplane` can `fly`.
- Default implementation: If `ModelC` does not override `Airplane::fly`, it will have the inherited implementation automatically.

Separate default implementation from interface

To sever the connection between the *interface* of the virtual function and its *default implementation*:

```
class Airplane {  
public:  
    virtual void fly(const Airport &destination) = 0; // pure virtual  
    // ...  
protected:  
    void defaultFly(const Airport &destination) {  
        // Default code for flying an airplane to the given destination.  
    }  
};
```

- The pure virtual function `fly` provides the **interface**: Every derived class can `fly`.
- The **default implementation** is written in `defaultFly`.

Separate default implementation from interface

If `ModelA` and `ModelB` want to adopt the default way of flying, they simply make a call to `defaultFly`.

```
class ModelA : public Airplane {
public:
    virtual void fly(const Airport &destination) {
        defaultFly(destination);
    }
    // ...
};
class ModelB : public Airplane {
public:
    virtual void fly(const Airport &destination) {
        defaultFly(destination);
    }
    // ...
};
```

Separate default implementation from interface

For `ModelC` :

- Since `Airplane::fly` is pure virtual, `ModelC` must define its own version of `fly` .
- If it **does** want to use the default implementation, it **must say it explicitly** by making a call to `defaultFly` .

```
class ModelC : public Airplane {  
public:  
    virtual void fly(const Airport &destination) {  
        // The "Model C way" of flying.  
        // Without the definition of this function, `ModelC` remains abstract,  
        // which does not compile if we create an object of such type.  
    }  
};
```

Still not satisfactory?

Some people object to the idea of having separate functions for providing the interface and the default implementation, such as `fly` and `defaultFly` above.

- For one thing, it pollutes the class namespace with closely related function names.
 - This really matters, especially in complicated projects. Extra mental effort might be required to distinguish the meaning of overly similar names.

Read the rest part of *Effective C++* Item 34 for another solution to this problem.

Inheritance of interface vs inheritance of implementation

We have come to the conclusion that

- Pure virtual functions specify **inheritance of interface** only.
- Simple (impure) virtual functions specify **inheritance of interface + a default implementation**.
 - The default implementation can be overridden.

Moreover, non-virtual functions specify **inheritance of interface + a mandatory implementation**.

Note: In public inheritance, *interfaces are always inherited*.

Summary

Pure virtual function and abstract class

- A pure virtual function is a virtual function declared `= 0`.
 - Call to a pure virtual function is not allowed. ¹
 - Pure virtual functions define the interfaces and force the derived classes to override it.
- A class that has a pure virtual function is an abstract class.
 - We cannot create an object of an abstract class type.
 - Abstract classes are often used to represent abstract, general concepts.

Summary

Public inheritance models the "is-a" relationship.

- Everything that applies to base classes must also apply to derived classes.
- The "Birds can fly, and a penguin is a bird" example.
- The "A square is a rectangle" example.

Summary

Inheritance of interface vs inheritance of implementation

- In public inheritance, interfaces are always inherited.
- Pure virtual functions: inheritance of **interface** only.
- Simple (impure) virtual functions: inheritance of **interface + a default implementation**.
 - The default implementation can be overridden.
- non-virtual functions: inheritance of **interface + a mandatory implementation**.

Notes

¹ A pure virtual function can have a definition. In that case, it can be called via the syntax `ClassName::functionName(args)`, not via a virtual function call (dynamic binding).

In some cases, we want a class to be made abstract, but it does not have any pure virtual function. A possible workaround is to declare the destructor to be pure virtual, and then provide a definition for it:

```
struct Foo {  
    virtual ~Foo() = 0;  
};  
Foo::~~Foo() = default; // Provide a definition outside the class.
```

The "another solution" mentioned in page 36 is also related to this.