CS100 Lecture 21

Inheritance and Polymorphism II

Contents

- Pure `virtual` functions and abstract classes
- Public inheritance for the "is-a" relationship (*Effective* C++ Item 32)
- Inheritance of interface vs. inheritance of implementation (*Effective* C++ Item 34)

Pure `virtual` functions and abstract classes

Shapes

Define classes to represent different shapes: Rectangle, Triangle, Circle, ...

Suppose we want to draw *different* shapes using a *single* function:

```cpp
void drawShapes(ScreenHandle &screen,
                const std::vector<std::shared_ptr<Shape>> &shapes) {
  for (const auto &shape : shapes)
    shape->draw(screen);
} // Pointers in `shapes` can point to objects of different shape classes.
```

and print their information using a *single* function:

```cpp
void printShapeInfo(const Shape &shape) {
  std::cout << "Area: " << shape.area()
            << "Perimeter: " << shape.perimeter() << std::endl;
} // `shape` can be bound to objects of different shape classes.
```

What is the relationship between `Shape` and other shape classes?

Shapes

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

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

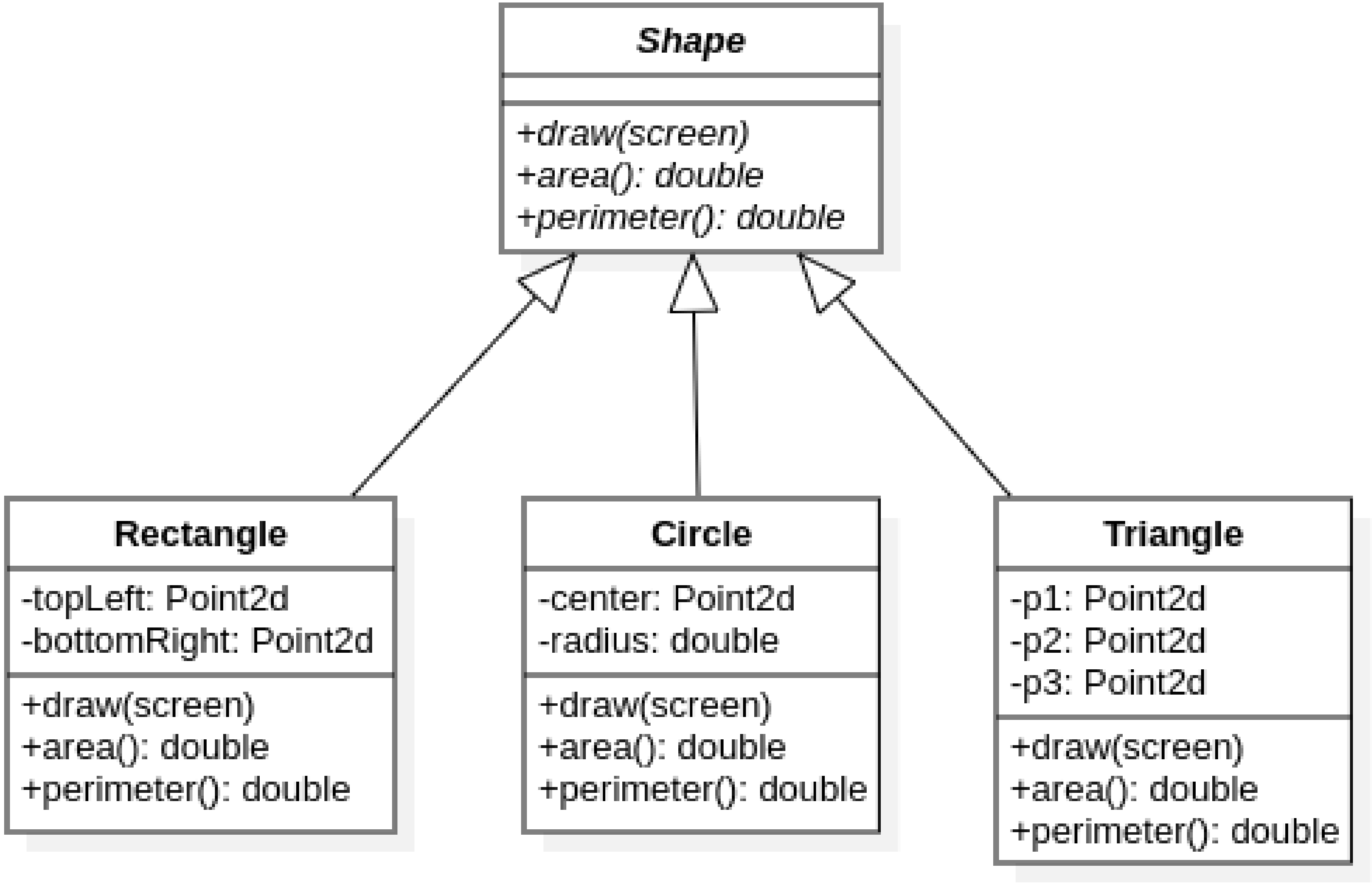
Different shape classes should define their own `draw`, `area` and `perimeter`, so these functions should be `virtual` in `Shape` (and thus can be overridden in the subclasses).

Shapes

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

public:
  Rectangle(const Point2d &tl, const Point2d &br)
      : mTopLeft(tl), mBottomRight(br) {} // call the default ctor of `Shape`
  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 provide required interfaces (declarations of functions that any shape class needs to support)**.

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 a 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.

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

Make the interface robust, not error-prone

Besides declaring `Shape::draw`, `Shape::area`, `Shape::perimeter` as pure `virtual`, other possible designs can be:

```cpp
class Shape {
public:
  virtual double area() const {
    return 0; // Is this good?
  }
};
```

```cpp
class Shape {
public:
  virtual double area() const {
    throw std::logic_error{"area() called on Shape!"}; // What about this?
  }
};
```

Make the interface robust, not error-prone

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

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

Make the interface robust, not error-prone

```cpp
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 at run-time.

However, **a good design should make errors fail to compile**, which can be achieved by declaring `Shape::area` as pure `virtual`.

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

Polymorphism (多态)

Polymorphism: The same function name can invoke different behaviors in different calling contexts.

- Run-time polymorphism: **dynamic binding** (contexts: types of objects being referred to by a base class pointer or reference on which the function is called).
- Compile-time polymorphism: **function overloading** (contexts: number or types of the function arguments, or `const`ness of objects on which the funciton is called).

Run-time polymorphism:

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

Compile-time polymorphism:

```cpp
struct Rectangle {
  void draw() const;
  void draw(const std::string &str) const;
};
Rectangle r;
r.draw();
r.draw("rectangle");
```

Public inheritance for the "is-a" relationship

Effective C++ Item 32

Public inheritance: The "is-a" relationship

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

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

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

- However, if you need an object of type `Sub` somewhere, an object of type `Base` can not be used.

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 entrolled in a particular school, but a person may not.

The notion of a person is **more general** than 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!
```

Not every bird can fly

In general, only birds with the flying ability can fly.

- There are several types of non-flying birds.

Maybe the following hierarchy models the reality much better?

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

Not every bird can fly

Maybe the following hierarchy models the reality much better?

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

- **Not necessary.** 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 reporting a runtime error?

To prevent the attempt to make penguins fly, another way is to redefine `Penguin::fly` to report an error at run-time:

```
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 reporting a runtime error?

```cpp
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!"); }
};
```

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."**

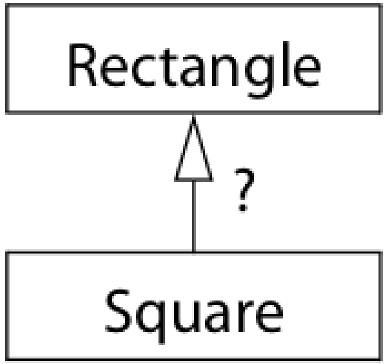
To actually express the constraint "Penguins can't fly", you should **prevent the attempt from compiling**, which can be achieved by removing `Penguin::fly`.

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

[Best practice] <u>Good interfaces prevent invalid code at **compile-time**.</u>

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 public inheritance suitable here?

Public inheritance models "is-a", which asserts that **everything that applies to base classes must also apply to subclasses**, because every subclass object is a base class object.

However, something applicable to a rectangle (i.e., height is independent of width) is not applicable to a square whose height and width must be the same!

Thus, public inheritance is not suitable for modeling the "is-a" relationship bewteen rectangle and square.

Public inheritance: The "is a" relationship

You need to be careful when using public inheritance to model "is a" in practice.

- To decide if public inheritance should be used, ask: **Does everything that applies to base classes are also applicable to subclasses?**

Not all "is-a" relationships can be naively modeled with public inheritance:

- "A penguin is a bird": something applicable to a bird (flying) is not applicable to a penguin.
- "A square is a rectangle": something applicable to a rectangle (independence of height and width) is not applicable to a square.

Inheritance of interface vs. inheritance of implementation

Effective C++ Item 34

Example: Airplanes for XYZ Airlines

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

```cpp
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 code repetition in `ModelA` and `ModelB` classes.

Example: Airplanes for XYZ Airlines

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

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

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

This surely leads to a disaster:

```cpp
auto pc = std::make_unique<ModelC>();
pc->fly(PVG); // No! Attempt 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 needs the behavior**.

By defining an impure virtual function in a base class, we have the subclass 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 cut off the connection between the *interface* 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 an independent function `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`.

```cpp
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.

- 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

Pure virtual functions specify **inheritance of interface** only.

- The base class tells the subclass: "You must support these functions, but I don't know how to implement them".

Impure virtual functions specify **inheritance of interface + default implementation**.

- The base class tells the subclass: "You must support these functions, and you can redefine (override) them or use the default implementations directly".

Non-virtual functions specify **inheritance of interface + mandatory implementation**.

- The base class tells the subclass: "You must support these functions and use my implementations".

Summary

Pure `virtual` functions and abstract classes

- A pure `virtual` function is a `virtual` function declared `= 0`.
  - Call to a pure `virtual` function is not allowed. [1]
  - Pure `virtual` functions provide interfaces and let the subclasses implement them.
- 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 subclasses.
- The "A bird can fly, and a penguin is a bird" example.
- The "A square is a rectangle" example.

Summary

Inheritance of interface vs. inheritance of implementation

- Pure virtual functions: inheritance of **interface** only.
- Impure virtual functions: inheritance of **interface + default implementation**.
- Non-virtual functions: inheritance of **interface + mandatory implementation**.
- In public inheritance, interfaces are always inherited.

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 37 is also related to this.