CS101A(H) Week3

Inheritance and Polymorphism

Contents

Inheritance

Example: An item for sale

```cpp
class Item {
  std::string m_name;
  double m_price = 0.0;
public:
  Item() = default;
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
  const auto &getName() const { return m_name; }
  auto netPrice(int cnt) const {
    return cnt * m_price;
  }
};
```

Defining a subclass

A discounted item **is an** item, and has more information:

- `std::size_t m_minQuantity;`
- `double m_discount;`

The net price for such an item is

$$\mathrm{netPrice}(n) = \begin{cases} n \cdot \mathrm{price}, & \text{if } n < \mathrm{minQuantity}, \\ n \cdot \mathrm{discount} \cdot \mathrm{price}, & \text{otherwise.} \end{cases}$$

Defining a subclass

Use **inheritance** to model the "is-a" relationship:

- A discounted item **is an** item.

```
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;
public:
  // constructors
  // netPrice
};
```

`protected` members

A `protected` member is private, except that it is accessible in subclasses.

- `m_price` needs to be `protected`, of course.
- Should `m_name` be `protected` or `private`?
  - `private` is ok if the subclass does not modify it. It is accessible through the public `getName` interface.
  - `protected` is also reasonable.

`protected` members

```cpp
class Item {
  std::string m_name;
protected:
  double m_price = 0.0;
public:
  Item() = default;
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
  const auto &getName() const { return m_name; }
  auto netPrice(int cnt) const {
    return cnt * m_price;
  }
};
```

Inheritance

By defining `DiscountedItem` to be a subclass of `Item`, **every `DiscountedItem` object contains a subobject of type `Item`.**

- Every data member and member function, except the ctors and dtors, is inherited, **no matter what access level they have**.

What can be inferred from this?

- A constructor of `DiscountedItem` must first initialize the base class subobject by calling a constructor of `Item`'s.
- The destructor of `DiscountedItem` must call the destructor of `Item` after having destroyed its own members (`m_minQuantity` and `m_discount`).
- `sizeof(Derived) ⩾ sizeof(Base)`

Inheritance

Key points of inheritance:

- Every object of the derived class (subclass) contains a base class subobject.
- Inheritance should not break the encapsulation of the base class.
  - e.g. To initialize the base class subobject, **we must call a constructor of the base class**. It is not allowed to initialize data members of the base class subobject directly.

Constructor of `DiscountedItem`

```cpp
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double disc)
      : Item(name, price), m_minQuantity(minQ), m_discount(disc) {}
};
```

It is not allowed to write this:

```cpp
DiscountedItem(const std::string &name, double price,
               int minQ, double disc)
    : m_name(name), m_price(price), m_minQuantity(minQ), m_discount(disc) {}
```

Constructor of derived classes

Before the initialization of the derived class's own data members, the base class subobject **must** be initialized by having one of its ctors called.

- What if we don't call the base class's ctor explicitly?

  ```
  DiscountedItem(...)
    : /* ctor of Item is not called */ m_minQuantity(minQ), m_discount(d) {}
  ```

Constructor of derived classes

Before the initialization of the derived class's own data members, the base class subobject **must** be initialized by having one of its ctors called.

- What if we don't call the base class's ctor explicitly?
  - The default constructor of the base class is called.
  - If the base class is not default-constructible, an error.
- What does this constructor do?

  ```
  DiscountedItem() = default;
  ```

Constructor of derived classes

Before the initialization of the derived class's own data members, the base class subobject **must** be initialized by having one of its ctors called.

- What if we don't call the base class's ctor explicitly?
  - The default constructor of the base class is called.
  - If the base class is not default-constructible, an error.
- What does this constructor do?

  ```
  DiscountedItem() = default;
  ```

  - Calls `Item::Item()` to default-initialize the base class subobject before initializing `m_minQuantity` and `m_discount`.

Constructor of derived classes

In the following code, does the constructor of `DiscountedItem` compile?

```cpp
class Item {
  // ...
public:
  // Since `Item` has a user-declared constructor, it does not have
  // a default constructor.
  Item(const std::string &name, double p) : m_name(name), m_price(p) {}
};
class DiscountedItem : public Item {
  // ...
public:
  DiscountedItem(const std::string &name, double p, int mq, double disc)
  // Before entering the function body, `Item::Item()` is called ⟶ Error!
  { /* ... */ }
};
```

[Best practice] Use constructor initializer lists whenever possible.

Dynamic binding

Upcasting

If `D` is a subclass of `B`:

- A `B*` can point to a `D`, and
- A `B&` can be bound to a `D`.

```
DiscountedItem di = someValue();
Item &ir = di; // correct
Item *ip = &di; // correct
```

Reason: The **is-a** relationship! A `D` **is a** `B`.

But on such references or pointers, only the members of `B` can be accessed.

Upcasting: Example

```cpp
void printItemName(const Item &item) {
  std::cout << "Name: " << item.getName() << std::endl;
}
DiscountedItem di("A", 10, 2, 0.8);
Item i("B", 15);
printItemName(i); // "Name: B"
printItemName(di); // "Name: A"
```

`const Item &item` can be bound to either an `Item` or a `DiscountedItem`.

Static type and dynamic type

- **static type** of an expression: The type known at compile-time.
- **dynamic type** of an expression: The real type of the object that the expression is representing. This is known at run-time.

```cpp
void printItemName(const Item &item) {
  std::cout << "Name: " << item.getName() << std::endl;
}
```

The static type of the expression `item` is `const Item`, but its dynamic type is not known until run-time. (It may be `const Item` or `const DiscountedItem`.)

`virtual` functions

`Item` and `DiscountedItem` have different ways of computing the net price.

```cpp
void printItemInfo(const Item &item) {
  std::cout << "Name: " << item.getName()
            << ", price: " << item.netPrice(1) << std::endl;
}
```

- Which `netPrice` should be called?
- How do we define two different `netPrice`s?

`virtual` functions

```cpp
class Item {
public:
  virtual double netPrice(int cnt) const {
    return m_price * cnt;
  }
  // other members
};
class DiscountedItem : public Item {
public:
  double netPrice(int cnt) const override {
    return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
  }
  // other members
};
```

Note: `auto` cannot be used to deduce the return type of `virtual` functions.

Dynamic binding

```cpp
void printItemInfo(const Item &item) {
  std::cout << "Name: " << item.getName()
            << ", price: " << item.netPrice(1) << std::endl;
}
```

The dynamic type of `item` is determined at run-time.

Since `netPrice` is a `virtual` function, which version is called is also determined at run-time:

- If the dynamic type of `item` is `Item`, it calls `Item::netPrice`.
- If the dynamic type of `item` is `DiscountedItem`, it calls `DiscountedItem::netPrice`.

late binding, or **dynamic binding**

`virtual`-`override`

To **override** (覆盖/覆写) a `virtual` function,

- The function parameter list must be the same as that of the base class's version.
- The return type should be **identical to** (or ***covariant with***) that of the corresponding function in the base class.
- **The `const`ness should be the same!**

To make sure you are truly overriding the `virtual` function (instead of making a overloaded version), use the `override` keyword.

*** Not to be confused with "overloading"（重载）.**

`virtual`-`override`

An overriding function is also `virtual`, even if not explicitly declared.

```
class DiscountedItem : public Item {
  virtual double netPrice(int cnt) const override; // correct, explicitly virtual
};
class DiscountedItem : public Item {
  double netPrice(int cnt) const; // also correct, but not recommended
};
```

The `override` keyword lets the compiler check and report if the function is not truly overriding.

[Best practice] To override a virtual function, write the `override` keyword explicitly. The `virtual` keyword can be omitted.

`virtual` destructors

```
Item *ip = nullptr;
if (some_condition)
  ip = new Item(/* ... */);
else
  ip = new DiscountedItem(/* ... */);
// ...
delete ip;
```

Whose destructor should be called?

- Only looking at the static type of `*ip` is not enough.
- It needs to be determined at run-time!
- **To use dynamic binding correctly, you almost always need a `virtual` destructor.**

`virtual` destructors

```
Item *ip = nullptr;
if (some_condition)
  ip = new Item(/* ... */);
else
  ip = new DiscountedItem(/* ... */);
// ...
delete ip;
```

- The implicitly-defined (compiler-generated) destructor is **non-** `virtual`, but we can explicitly require a `virtual` one:

  ```
  virtual ~Item() = default;
  ```

- If the dtor of the base class is `virtual`, the compiler-generated dtor for the derived class is also `virtual`.

(Almost) completed `Item` and `DiscountedItem`

```cpp
class Item {
  std::string m_name;

protected:
  double m_price = 0.0;

public:
  Item() = default;
  Item(const std::string &name, double price) : m_name(name), m_price(price) {}
  const auto &getName() const { return name; }
  virtual double net_price(int n) const {
    return n * price;
  }
  virtual ~Item() = default;
};
```

(Almost) completed `Item` and `DiscountedItem`

```cpp
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;

public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double disc)
      : Item(name, price), m_minQuantity(minQ), m_discount(disc) {}
  double netPrice(int cnt) const override {
    return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
  }
};
```

Copy-control

Remember to copy/move the base subobject! One possible way:

```cpp
class Derived : public Base {
public:
  Derived(const Derived &other)
      : Base(other), /* Derived's own members */ { /* ... */ }
  Derived &operator=(const Derived &other) {
    Base::operator=(other); // call Base's operator= explicitly
    // copy Derived's own members
    return *this;
  }
  // ...
};
```

Why `Base(other)` and `Base::operator=(other)` work?

- The parameter type is `const Base &`, which can be bound to a `Derived` object.

Synthesized copy-control members

Guess!

- What are the behaviors of the compiler-generated copy-control members?
- In what cases will they be `delete`d?

Synthesized copy-control members

Remeber that the base class's subobject is always handled first.

These rules are quite natural:

- What are the behaviors of the compiler-generated copy-control members?
  - First, it calls the base class's corresponding copy-control member.
  - Then, it performs the corresponding operation on the derived class's own data members.
- In what cases will they be `delete`d?
  - If the base class's corresponding copy-control member is not accessible (e.g. non-existent, or `private`),
  - or if any of the data members' corresponding copy-control member is not accessible.

Slicing

Dynamic binding only happens on references or pointers to base class.

```
DiscountedItem di("A", 10, 2, 0.8);
Item i = di; // What happens?
auto x = i.netPrice(3); // Which netPrice?
```

`Item i = di;` calls the **copy constructor of** `Item`

- but `Item`'s copy constructor handles only the base part.
- So `DiscountedItem`'s own members are **ignored**, or **"sliced down"**.
- `i.netPrice(3)` calls `Item::netPrice`.

Downcasting

```cpp
Base *bp = new Derived{};
```

If we only have a `Base` pointer, but we are quite sure that it points to a `Derived` object

- Accessing the members of `Derived` through `bp` is not allowed.
- How can we perform a **"downcasting"**?

Polymorphic class

A class is said to be **polymorphic** if it has (declares or inherits) at least one virtual function.

- Either a `virtual` normal member function or a `virtual` dtor is ok.

If a class is polymorphic, all classes derived from it are polymorphic.

- There is no way to "refuse" to inherit any member functions, so `virtual` member functions must be inherited.
- The dtor must be `virtual` if the dtor of the base class is `virtual`.

Downcasting: For polymorphic class only

`dynamic_cast<Target>(expr)`.

```
Base *bp = new Derived{};
Derived *dp = dynamic_cast<Derived *>(bp);
Derived &dr = dynamic_cast<Derived &>(*bp);
```

- `Target` must be a **reference** or a **pointer** type.
- `dynamic_cast` will perform **runtime type identification (RTTI)** to check the dynamic type of the expression.
  - If the dynamic type is `Derived`, or a derived class (direct or indirect) of `Derived`, the downcasting succeeds.
  - Otherwise, the downcasting fails. If `Target` is a pointer, returns a null pointer. If `Target` is a reference, throws an exception `std::bad_cast`.

`dynamic_cast` can be very slow

`dynamic_cast` performs a runtime **check** to see whether the downcasting should succeed, which uses runtime type information.

This is **much slower** than other types of casting, e.g. `const_cast`, or arithmetic conversions.

[Best practice] Avoid `dynamic_cast` whenever possible.

Guaranteed successful downcasting: Use `static_cast`.

If the downcasting is guaranteed to be successful, you may use `static_cast`

```
auto dp = static_cast<Derived *>(bp); // quicker than dynamic_cast,
// but performs no checks. If the dynamic type is not Derived, UB.
```

Avoiding `dynamic_cast`

Typical abuse of `dynamic_cast`:

```
struct A {
  virtual ~A() {}
};
struct B : A {};
struct C : A {};
std::string getType(const A *ap) {
  if (dynamic_cast<const B *>(ap))
    return "B";
  else if (dynamic_cast<const C *>(ap))
    return "C";
  else
    return "A";
}
```

Click here to see how large and slow the generated code is:
https://godbolt.org/z/3367efGd7

Avoiding `dynamic_cast`

Use a group of `virtual` functions!

```
struct A {
  virtual ~A() {}
  virtual std::string name() const {
    return "A";
  }
};
struct B : A {
  std::string name()const override{
    return "B";
  }
};
struct C : A {
  std::string name()const override{
    return "C";
  }
};
```

Avoiding `dynamic_cast`

Use a group of `virtual` functions!

```
auto getType(const A *ap) {
  return ap->name();
}
```

- This time: https://godbolt.org/z/KosbcaGnT

  The generated code is much simpler!

Abstract base class

Shapes

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

Suppose we want to draw things like this:

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

and print information:

```cpp
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.

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

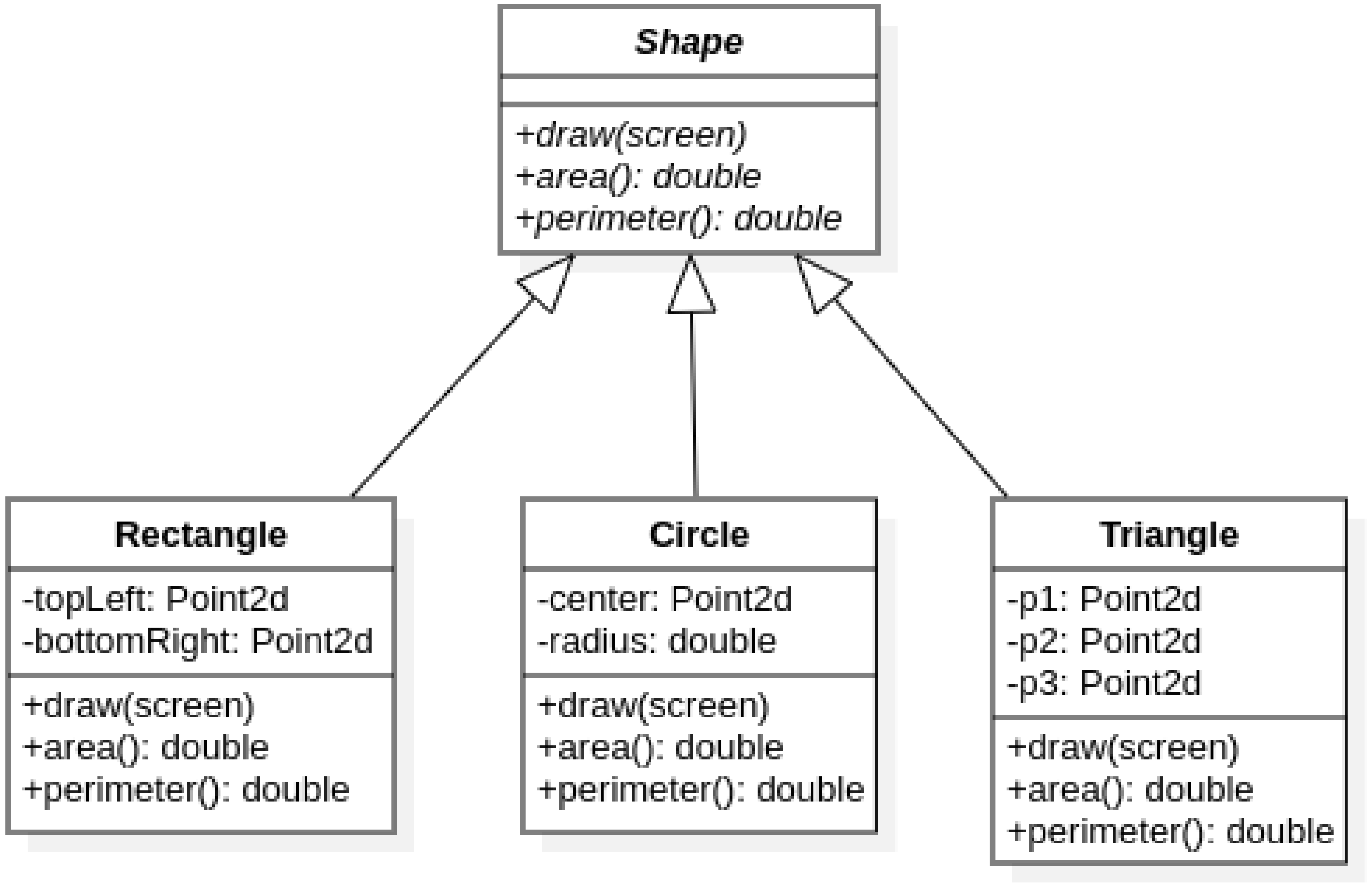
Different shapes should define their own `draw`, `area` and `perimeter`, so these functions should be `virtual`.

Shapes

```cpp
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.
- 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.

```cpp
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?

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

What about this?

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

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***!

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.

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

[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 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:

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

Compile-time polymorphism:

```cpp
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 entrolled 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**.

```cpp
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:

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

```cpp
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?

```cpp
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?

```cpp
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?

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

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."**

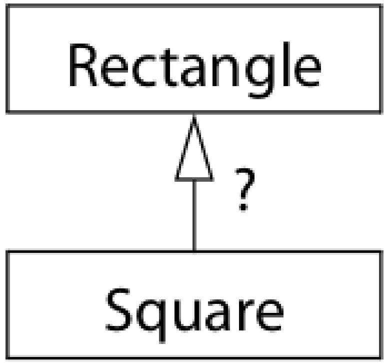
To actually express the constraint "Penguins can't fly", you should prevent the attempt from **compiling**.

```cpp
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.
  // ...
};
```

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!

```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! 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`.

```cpp
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*.