**Inheritance and Object-Oriented Design**

* Public inheritance means “is-a,”
* A virtual function means “interface must be inherited,” while a non-virtual function means “both interface and implementation must be inherited.

Failing to distinguish between these meanings has caused C++ programmers considerable grief.

**Item 32: Make sure public inheritance models “is-a.”**

If you write that class D (“Derived”) publicly inherits from class B(“Base”), It means every object of type D is also an object of type B, but not vice versa. You are saying that B represents a more general concept than D, that D represents a more specialized concept than B. You are asserting that anywhere an object of type B can be used, an object of type D can be used just as well, because every object of type D is an object of type B. On the other hand, if you need an object of type D, an object of type B will not do: every D is-a B, but not vice versa. Consider this example:

class Person { ... };

class Student: public Person { ... }; //Every student is a person, but not every person //is a student

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

void eat(const Person& p); // anyone can eat

void study(const Student& s); // only students study

Person p; // p is a Person

Student s; // s is a Student

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

But sometimes it’s very difficult to choose the right relationship. (public inheritance is completely different then private and protected one.) For example, it is a fact that a penguin is a bird, and it is a fact that birds can fly.

class Bird {

public:

virtual void fly(); // birds can fly

...

};

class Penguin: public Bird { // penguins are birds

...

};

Suddenly we are in trouble, because this hierarchy says that penguins can fly, which we know is not true. When we say that birds can fly, we don’t mean that all types of birds can fly, only that, in general, birds have the ability to fly. If we were more precise, we’d recognize that there are several types of non-flying birds, and we would come up with the following hierarchy:

class Bird {

... // no fly function is declared

};

class FlyingBird: public Bird {

public:

virtual void fly();

...

};

class Penguin: public Bird {

... // no fly function is declared

}

There is no one ideal design for all software. The best design depends on what the system is expected to do, both now and in the future.

Suppose in school we taught like this “All birds can fly, penguins are birds, penguins can’t fly, uh oh” problem. That is to redefine the fly function for penguins so that it generates a runtime error:

void error(const std::string& msg); // defined elsewhere

class Penguin: public Bird {

public:

virtual void fly() { error("Attempt to make a penguin fly!");

}

...

};

This does not say, “Penguins can’t fly.” This says, “Penguins can fly, but it’s an error for them to actually try to do it.” It’s an error for penguins to actually try to fly,” can be detected only at runtime.

But if we define our interface like this:

class Bird {

... // no fly function is declared

class Penguin: public Bird {

... // no fly function is declared

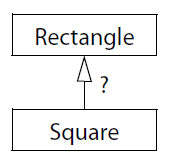
};

Penguin p;

p.fly(); // error!

Now the both error versions are different. Good interfaces prevent invalid code from compiling, so above design rejects penguin flight attempts during compilation to the one that detects them only at runtime.

Another example: class Square publicly inherit from class Rectangle? Everybody knows that a square is a rectangle, but generally not vice versa.



class Rectangle {

public:

virtual void setHeight(int newHeight);

virtual void setWidth(int newWidth);

virtual int height() const; // return current values

virtual int width() const;

...

};

void makeBigger(Rectangle& r) // function to increase r’s area

{

int oldHeight = r.height();

r.setWidth(r.width() + 10); // add 10 to r’s width

assert(r.height() == oldHeight); // assert that r’s

}

class Square: public Rectangle { ... };

Square s;

assert(s.width() == s.height()); // this must be true for all squares

makeBigger(s); // by inheritance, s is-a Rectangle,

// so, we can increase its area.

assert(s.width() == s.height()); // this must still be true for all squares.

* Before calling makeBigger, s’s height is the same as its width.
* Inside makeBigger, s’s width is changed, but its height is not.
* After returning from makeBigger, s’s height & width are unequal.

The fundamental difficulty in this case is that something applicable to a rectangle (its width may be modified independently of its height) is not applicable to a square (its width and height must be the same). But public inheritance asserts that everything that applies to base class objects everything! also applies to derived class objects.

The is-a relationship is not the only one that can exist between classes. Two other common inter-class relationships are “has-a” and “is-implemented-in-terms-of.”

**Things to Remember**

✦ 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.

**Item 33: Avoid hiding inherited names.**

C++’s name-hiding rules do just that: hide names. Whether the names correspond to the same or different types is immaterial. In this case, a double named x hides an int named x.

int x; // global variable

void someFunc()

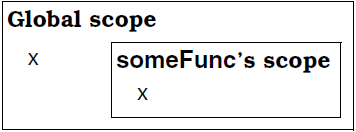
{

double x; // local variable

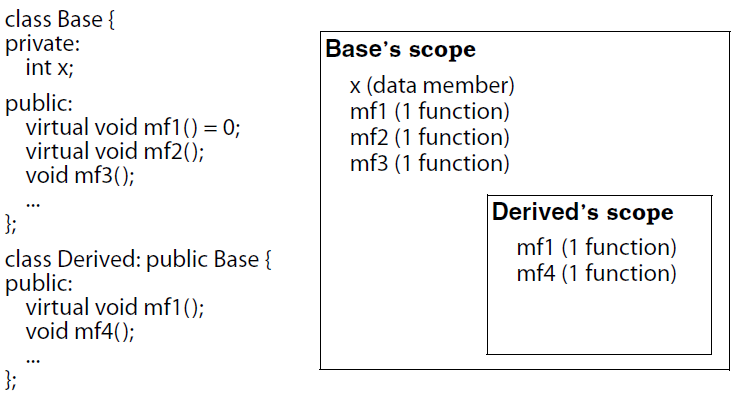
std::cin >> x; // read a new value for local x

}

the statement reading into x refers to the local variable x instead of the global variable x, because names in inner scopes hide (“shadow”) names in outer scopes.



Enter inheritance. We know that when we’re inside a derived class member function and we refer to something in a base class, compilers can find what we’re referring to because derived classes inherit the things declared in base classes. The way that works is that the scope of a derived class is nested inside its base class’s scope. For example



Suppose, mf4 in the derived class is implemented, like this:

void Derived::mf4()

{

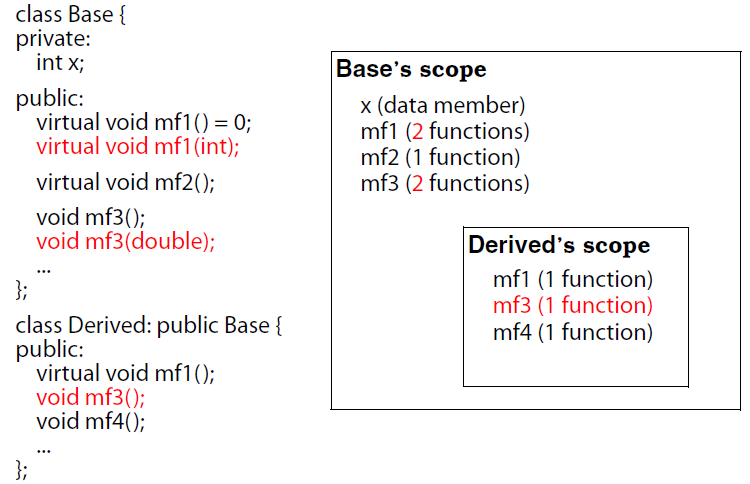
...

mf2();

}

When compilers see the use of the name mf2 here, they must figure out what it refers to. Starts searching scopes for a declaration of something named mf2. First, they look in the local scope (that of mf4), but they find no declaration for anything called mf2. They then search the containing scope, that of the class Derived. They still find nothing named mf2, so they move on to the next containing scope, that of the base class. There they find something named mf2, so the search stops.

Consider the previous example again, except this time let’s overload mf1 and mf3, and let’s add a version of mf3 to Derived.



It would be little bit surprising. The scope-based name hiding rule hasn’t changed, so all functions named mf1 and mf3 in the base class are hidden by the functions named mf1 and mf3 in the derived class. From the perspective of name lookup, Base::mf1 and Base::mf3 are no longer inherited by Derived!

This applies even though the functions in the base and derived classes take different parameter types, and it also applies regardless of whether the functions are virtual or non-virtual.

Derived d;

int x;

...

d.mf1(); // fine, calls Derived::mf1

d.mf1(x); // error! Derived::mf1 hides Base::mf1

d.mf2(); // fine, calls Base::mf2

d.mf3(); // fine, calls Derived::mf3

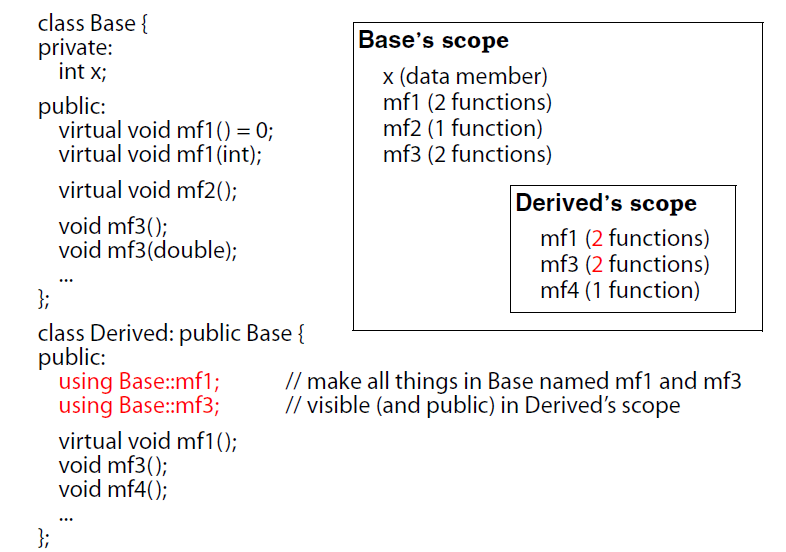
d.mf3(x); // error! Derived::mf3 hides Base::mf3

At the beginning of this Item, the double x in the function someFunc hides the int x at global scope, here the function mf3 in Derived hides a Base function named mf3 that has a different type.

*The notion behind this behavior is that, it prevents from accidentally inheriting overloads from base classes when we create a new derived class in application framework.*

Unfortunately, we want to inherit the overloads. Because using public inheritance and don’t inherit the overloads, that means we are violating the is-a relationship between base class.

We can do it with using declarations:



Now inheritance will work as expected.

*This means, if we inherit from a base class with overloaded functions and you want to redefine or override only some of them, need to include a using declaration for each name, otherwise it will be hidden.*

Under private inheritance things are little bit different. For example, suppose Derived privately inherits from Base, and the only version of mf1 that Derived wants which taking no parameters. A using declaration won’t work here, *because a using declaration makes all inherited functions with a given name visible in the derived class*. No, this is a case for a different technique, namely, a simple forwarding function:

class Base {

public:

virtual void mf1() = 0;

virtual void mf1(int);

... // as before

};

class Derived: private Base {

public:

virtual void mf1()

{ Base::mf1(); }

...

};

...

Derived d;

int x;

d.mf1(); // fine, calls Derived::mf1

d.mf1(x); // error! Base::mf1() is hidden

**Things to Remember**

✦ Names in derived classes hide names in base classes. Under public inheritance, this is

never desirable.

✦ To make hidden names visible again, employ using declarations (public inheritance) or

forwarding functions (private inheritance).

**Item 34: Differentiate between inheritance of interface and inheritance of implementation.**

We have two different things: inheritance of function interfaces and inheritance of function implementations.

As a class designer, sometimes:

* We want derived classes to inherit only the interface (declaration) of a member function.
* We want derived classes to inherit both a function’s interface and implementation but allow them to override the implementation they inherit.
* We want derived classes to inherit a function’s interface and implementation without allowing them to override anything.

Let’s see one by one. Consider a class hierarchy for representing geometric shapes in a graphics application:

class Shape {

public:

virtual void draw() const = 0;

virtual void error(const std::string& msg);

int objectID() const;

...

};

class Rectangle: public Shape { ... };

class Ellipse: public Shape { ... };

Three functions are declared in the Shape class. The first, draw, draws the current object on an implicit display. The second, error, is called when an error needs to be reported. The third, objectID, returns a unique integer identifier for the current object. Each function is declared in a different way: draw is a pure virtual function; error is a simple virtual function; and objectID is a non-virtual function.

*Consider first the pure virtual function draw:*

Pure virtual function draw makes, clients cannot create instances of the Shape class, only of classes derived from it.

The two most salient features of pure virtual functions are:

* They must be redeclared by any concrete class that inherits them, and
* Typically have no definition in abstract classes. Put these two characteristics together

■ The purpose of declaring a pure virtual function is to have derived classes inherit a function interface only.

This makes perfect sense for the Shape::draw function, because it is a reasonable demand that all Shape objects must be drawable, but the Shape class can provide no reasonable default implementation for that function. The algorithm for drawing an ellipse is very different from the algorithm for drawing a rectangle.

The declaration of Shape::draw says to designers of concrete derived classes, “You must provide a draw function, but I have no idea how you’re going to implement it.”

*Note: it is possible to provide a definition for a pure virtual function (default implementation, that behave similar to simple virtual function).*

Shape \*ps = new Shape; // error! Shape is abstract

Shape \*ps1 = new Rectangle; // fine

ps1->draw(); // calls Rectangle::draw

Shape \*ps2 = new Ellipse; // fine

ps2->draw(); // calls Ellipse::draw

ps1->Shape::draw(); // calls Shape::draw, only if Shape::draw() has definition

ps2->Shape::draw(); // calls Shape::draw, only if Shape::draw() has definition

*Simple virtual functions* are bit different from the pure virtual. As usual, derived classes inherit the interface of the function, but simple virtual functions provide an implementation that derived classes may override.

■ The purpose of declaring a simple virtual function is to have derived classes inherit a function interface as well as a default implementation.

Consider the case of Shape::error:

class Shape {

public:

virtual void error(const std::string& msg);

...

};

The interface *error* says that every class must support a function to be called when an error is encountered, but each class is free to handle errors in whatever way it sees fit. If a class doesn’t want to do anything special, it can just fall back on the default error handling provided in the Shape class.

That is, the declaration of Shape::error says to designers of derived classes, “You’ve got to support an error function, but if you don’t want to write your own, you can fall back on the

default version in the Shape class.”

Another example:

Airline company lets says Delta, having two different model of Airplane.

class Airplane {

public:

virtual void fly(const Airport& destination);

...

};

void Airplane::fly(const Airport& destination)

{

//default code for flying an airplane to the given destination

}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };

This is a classic object-oriented design. Two classes share a common feature (the way they implement fly), so the common feature is moved into a base class, and the feature is inherited by the two classes (enforce reusability, less code maintenance).

Later Delta decides to acquire a new type of airplane, the Model C. The Model C differs in some ways from the Model A and the Model B. *It is flown differently*. So, developer inherited model C from Airplane but forgot to redefine the fly function. Then what’s happened next?

class ModelC: public Airplane {

... // no fly function is declared, forgotten by developer

};

Airplane \*pa = new ModelC;

...

pa->fly(PDX); // calls Airplane::fly!

This is a disaster. Because the fly() behavior of ModelC type airplane is completely different then default behavior which is defined in base class.

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. Fortunately, it’s easy to offer default behavior to derived classes but not give it to them unless they ask for it. Checkout the below code:

class Airplane {

public:

virtual void fly(const Airport& destination) = 0; //Now, fly() is pure virtual

...

protected:

void defaultFly(const Airport& destination);

};

void Airplane::defaultFly(const Airport& destination){

default code for flying an airplane to the given destination

}

The default implementation of fly() is present in the Airplane class, but now it’s in the form of an independent function, defaultFly(). Classes like ModelA and ModelB that want to use the default behavior simply make an inline call to defaultFly inside their body of fly.

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);

}

...

};

For the ModelC class, there is no possibility of accidentally inheriting the incorrect implementation of fly, because the pure virtual in Airplane forces ModelC to provide its own version of fly.

class ModelC: public Airplane {

public:

virtual void fly(const Airport& destination);

...

};

void ModelC::fly(const Airport& destination){

code for flying a ModelC airplane to the given destination

}

Still we may think, default implementation, such as fly and defaultFly pollutes the class namespace with a proliferation of closely related function names.

*By taking advantage of the fact that pure virtual functions must be redeclared in concrete derived classes, but they may also have implementations of their own.*

class Airplane {

public:

virtual void fly(const Airport& destination) = 0;

...

};

void Airplane::fly(const Airport& destination){ // an implementation of a pure virtual function

default code for flying an airplane to the given destination

}

class ModelA: public Airplane {

public:

virtual void fly(const Airport& destination){

Airplane::fly(destination);

}

...

};

class ModelB: public Airplane {

public:

virtual void fly(const Airport& destination){

Airplane::fly(destination);

}

...

};

class ModelC: public Airplane {

public:

virtual void fly(const Airport& destination){

code for flying a ModelC airplane to the given destination

}

...

};

Now, fly has been broken into its two fundamental components. Its declaration specifies its interface (which derived classes must use), while its definition specifies its default behavior (which derived classes may use, but only if they explicitly request it).

*Finally, we come to Shape’s non-virtual function, objectID:*

class Shape {

public:

int objectID() const;

...

};

When a member function is non-virtual, it’s not supposed to behave differently in derived classes. In fact, a non-virtual member function specifies an invariant over specialization, because it identifies behavior that is not supposed to change, no matter how specialized a derived class becomes.

■ The purpose of declaring a non-virtual function is to have derived classes inherit a function interface as well as a mandatory implementation.

Every Shape object has a function that yields an object identifier, and that object identifier is always computed the same way. And no derived class should try to change how it’s done.” Because a non-virtual function identifies an invariant over specialization, it should never be redefined in a derived class.

Some functions should not be redefinable in derived classes, and whenever that’s the case, you’ve got to say so by making those functions non-virtual.

**Things to Remember**

✦ Inheritance of interface is different from inheritance of implementation. Under public

inheritance, derived classes always inherit base class interfaces.

✦ Pure virtual functions specify inheritance of interface only.

✦ Simple (impure) virtual functions specify inheritance of interface plus inheritance of a

default implementation.

✦ Non-virtual functions specify inheritance of interface plus inheritance of a mandatory

implementation.

**Item 35: Consider alternatives to virtual functions.**

**Item 36: Never redefine an inherited non-virtual function.**

Consider the below example:

class Dog {

public:

void bark() {

std::cout << "Dog bark\n";

}

};

class SpacialDog :public Dog {

public:

void bark() {

std::cout << "spaical Dog bark\n";

}

};

SpacialDog sd;

Dog \*dp = &sd; //dp hold pointer of Spacial Dog

dp->bark(); //Called bark() thru pointer

SpacialDog \*sdp = &sd; //sdp hold pointer of Spacial Dog

sdp->bark(); //Called bark() thru pointer

In both cases invoking the member function bark() on the object sd. Because it’s the same function and the same object in both cases, it should behave the same way, right?

But output is:

Dog bark

Special Dog bark

class SpacialDog :public Dog {

public:

void bark() {

std::cout << "spaical Dog bark\n"; //hide Dog::Bark()

}

};

dp->bark(); //Call Dog::bark()

spd->bark(); //Call SpecialDog::bark()

The reason for this two-faced behavior is that non-virtual functions like Dog::bark and SpacialDog::bark are statically bound.

Virtual functions, on the other hand, are dynamically bound, so they don’t suffer from this problem. If bark() were a virtual function, a call to mf through either dp or spd would result in an invocation of SpecialDog::bark, because what dp and spd really point to is an object of type SpaicalDog.

If we are writing derived class SpeicalDog and redefine a non-virtual function bark() then derived class objects will likely exhibit inconsistent behavior.

As discussed in last item:

* Everything that applies to Base class objects also applies to Derived class objects, because every Derived object is-a Base object;
* Classes derived from Base must inherit both the interface and the implementation

of *bark*, because *bark* is non-virtual in B.

Now, if Derived class *specialDog* redefines *bark*, there is a contradiction in above design.

**Things to Remember**

✦ Never redefine an inherited non-virtual function.

**Item 37: Never redefine a function’s inherited default parameter value.**

There are only two kinds of functions we can inherit: virtual and non-virtual. However,

it’s always a mistake to redefine an inherited non-virtual function. so we can safely limit our discussion here to the situation in which we inherit a virtual function with a default parameter value.

*virtual functions are dynamically bound, but default parameter values are statically bound.*

class Shape { // a class for geometric shapes

public:

enum ShapeColor { Red, Green, Blue };

// all shapes must offer a function to draw themselves

virtual void draw(ShapeColor color = Red) const = 0;

...

};

class Rectangle: public Shape {

public:

// notice the different default parameter value — bad!

virtual void draw(ShapeColor color = Green) const;

...

};

class Circle: public Shape {

public:

virtual void draw(ShapeColor color) const;

...

};

Now consider these pointers:

Shape \*ps;

Shape \*pc = new Circle;

Shape \*pr = new Rectangle;

An object’s *dynamic type* is determined by the type of the object to which it currently refers. That is, its dynamic type indicates how it will behave. In the example above, pc’s dynamic type is Circle\*, and pr’s dynamic type is Rectangle\*. As for ps, it doesn’t really have a dynamic

type, because it doesn’t refer to any object (yet).

*But type of ps, pc and pr is shape statically.*

Dynamic types, as their name suggests, can change as a program runs, typically through assignments:

ps = pc; // ps’s dynamic type is now Circle\*

ps = pr; // ps’s dynamic type is now Rectangle\*

pc->draw(Shape::Red); // calls Circle::draw(Shape::Red)

pr->draw(Shape::Red); // calls Rectangle::draw(Shape::Red)

Now problem arise with below call:

pr->draw();

What do you think will it call Rectangle::draw(Shape::Green)? Ans: no

It calls Rectangle::draw(Shape::Red) but why???

virtual functions are dynamically bound, but default parameters are statically bound. That

means we may end up invoking a virtual function defined in a derived class but using a default parameter value from a base class.

In this case, pr’s dynamic type is Rectangle\*, so the Rectangle virtual function is called. Rectangle::draw, the default parameter value is Green. Because pr’s static type is Shape\*,

however, the default parameter value for this function call is taken from the Shape class, not the Rectangle class! The result is a call consisting of a strange and almost certainly unanticipated combination of the declarations for draw in both the Shape and Rectangle classes.

**Things to Remember**

✦ Never redefine an inherited default parameter value, because default parameter values are statically bound, while virtual functions — the only functions you should be redefining — are dynamically bound.