Inheritance — virtual functions

isocpp.org/wiki/faq/virtual-functions

Wiki Home > Inheritance — `virtual` functions
View

What is a “`virtual` member function”?

Virtual member functions are key to the object-oriented paradigm, such as making it easy for old code to call new code.

A `virtual` function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer. This allows algorithms in the base class to be replaced in the derived class, even if users don’t know about the derived class.

The derived class can either fully replace (“override”) the base class member function, or the derived class can partially replace (“augment”) the base class member function. The latter is accomplished by having the derived class member function call the base class member function, if desired.

Why are member functions not `virtual` by default?

Because many classes are not designed to be used as base classes. For example, see class complex.

Also, objects of a class with a virtual function require space needed by the virtual function call mechanism - typically one word per object. This overhead can be significant, and can get in the way of layout compatibility with data from other languages (e.g. C and Fortran).

See The Design and Evolution of C++ for more design rationale.

How can C++ achieve dynamic binding yet also static typing?

When you have a pointer to an object, the object may actually be of a class that is derived from the class of the pointer (e.g., a `Vehicle*` that is actually pointing to a `Car` object; this is called “polymorphism”). Thus there are two types: the (static) type of the pointer (`Vehicle` , in this case), and the (dynamic) type of the pointed-to object (`Car` , in this case).

Static typing means that the legality of a member function invocation is checked at the earliest possible moment: by the compiler at compile time. The compiler uses the static type of the pointer to determine whether the member function invocation is legal. If the type of the pointer can handle the member function, certainly the pointed-to object can handle it as well. E.g., if `Vehicle` has a certain member function, certainly `Car` also has that member function since `Car` is a kind-of `Vehicle` .

Dynamic binding means that the address of the code in a member function invocation is determined at the last possible moment: based on the dynamic type of the object at run time. It is called “dynamic binding” because the binding to the code that actually gets called is accomplished dynamically (at run time). Dynamic binding is a result of `virtual` functions.

What is a pure virtual function?

A pure virtual function is a function that must be overridden in a derived class and need not be defined. A virtual function is declared to be “pure” using the curious `=0` syntax. For example:

```
class Base {
public:
    void f1();      // not virtual
    virtual void f2();  // virtual, not pure
    virtual void f3() = 0;  // pure virtual
};
Base b; // error: pure virtual f3 not overridden
```

Here, `Base` is an abstract class (because it has a pure virtual function), so no objects of class `Base` can be directly created: `Base` is (explicitly) meant to be a base class. For example:

```
class Derived : public Base {
   // no f1: fine
   // no f2: fine, we inherit Base::f2
   void f3();
};
Derived d;  // ok: Derived::f3 overrides Base::f3
```

Abstract classes are immensely useful for defining interfaces. In fact, a class with no data and where all functions are pure virtual functions is often called an interface.

You can provide a definition for a pure virtual function:

```
Base::f3() { /* ... */ }
```

This is very occasionally useful (to provide some simple common implementation detail for derived classes), but `Base::f3()` must still be overridden in some derived class. If you don't override a pure virtual function in a derived class, that derived class becomes abstract:

```
class D2 : public Base {
   // no f1: fine
   // no f2: fine, we inherit Base::f2
   // no f3: fine, but D2 is therefore still abstract
};
D2 d;   // error: pure virtual Base::f3 not overridden
```

What's the difference between how `virtual` and non-`virtual` member functions are called?

Non-`virtual` member functions are resolved statically. That is, the member function is selected statically (at compile-time) based on the type of the pointer (or reference) to the object.

In contrast, `virtual` member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/reference to that object. This is called "dynamic binding." Most compilers use some variant of the following technique: if the object has one or more `virtual` functions, the compiler puts a hidden pointer in the object called a "virtual-pointer" or "v-pointer." This v-pointer points to a global table called the "virtual-table" or "v-table."

The compiler creates a v-table for each class that has at least one `virtual` function. For example, if class `Circle` has `virtual` functions for `draw()` and `move()` and `resize()`, there would be exactly one v-table associated with class `Circle`, even if there were a gazillion `Circle` objects, and the v-pointer of each of those `Circle` objects would point to the `Circle` v-table. The v-table itself has pointers to each of the virtual functions in the class. For example, the `Circle` v-table would have three pointers: a pointer to `Circle::draw()`, a pointer to `Circle::move()`, and a pointer to `Circle::resize()`.

During a dispatch of a `virtual` function, the run-time system follows the object's v-pointer to the class's v-table, then follows the appropriate slot in the v-table to the method code.

The space-cost overhead of the above technique is nominal: an extra pointer per object (but only for objects that will need to do dynamic binding), plus an extra pointer per method (but only for virtual methods). The time-cost overhead is also fairly nominal: compared to a normal function call, a `virtual` function call requires two extra fetches (one to get the value of the v-pointer, a second to get the address of the method). None of this runtime activity happens with non-`virtual` functions, since the compiler resolves non-`virtual` functions exclusively at compile-time based on the type of the pointer.

Note: the above discussion is simplified considerably, since it doesn't account for extra structural things like multiple inheritance, `virtual` inheritance, RTTI, etc., nor does it account for space/speed issues such as page faults, calling a function via a pointer-to-function, etc. If you want to know about those other things, please ask comp.lang.c++; PLEASE DO NOT SEND E-MAIL TO ME!

What happens in the hardware when I call a virtual function? How many layers of indirection are there? How much overhead is there?

This is a drill-down of the previous FAQ. The answer is entirely compiler-dependent, so your mileage may vary, but most C++ compilers use a scheme similar to the one presented here.

Let's work an example. Suppose class `Base` has 5 virtual functions: `virt0()` through `virt4()`.

```
// Your original C++ source code
class Base {
public:
  virtual arbitrary_return_type virt0( /*...arbitrary params...*/ );
  virtual arbitrary_return_type virt1( /*...arbitrary params...*/ );
  virtual arbitrary_return_type virt2( /*...arbitrary params...*/ );
  virtual arbitrary_return_type virt3( /*...arbitrary params...*/ );
  virtual arbitrary_return_type virt4( /*...arbitrary params...*/ );
  // ...
};
```

Step #1: the compiler builds a static table containing 5 function-pointers, burying that table into static memory somewhere. Many (not all) compilers define this table while compiling the .cpp that defines `Base`'s first non-inline virtual function. We call that table the v-table; let's pretend its technical name is `Base::__vtable`. If a function pointer fits into one machine word on the target hardware platform, `Base::__vtable` will end up consuming 5 hidden words of memory. Not 5 per instance, not 5 per function; just 5. It might look something like the following pseudo-code:

```
// Pseudo-code (not C++, not C) for a static table defined within file Base.cpp
// Pretend FunctionPtr is a generic pointer to a generic member function
// (Remember: this is pseudo-code, not C++ code)
FunctionPtr Base::__vtable[5] = {
  &Base::virt0, &Base::virt1, &Base::virt2, &Base::virt3, &Base::virt4
};
```

Step #2: the compiler adds a hidden pointer (typically also a machine-word) to each object of class `Base`. This is called the v-pointer. Think of this hidden pointer as a hidden data member, as if the compiler rewrites your class to something like this:

```
// Your original C++ source code
class Base {
public:
  // ...
  FunctionPtr* __vptr;  // Supplied by the compiler, hidden from the programmer
  // ...
};
```

Step #3: the compiler initializes `this->__vptr` within each constructor. The idea is to cause each object's v-pointer to point at its class's v-table, as if it adds the following instruction in each constructor's init-list:

```
Base::Base( /*...arbitrary params...*/ )
  : __vptr(&Base::__vtable[0])  // Supplied by the compiler, hidden from the programmer
  // ...
{
  // ...
}
```

Now let’s work out a derived class. Suppose your C++ code defines class `Der` that inherits from class `Base`. The compiler repeats steps #1 and #3 (but not #2). In step #1, the compiler creates a hidden v-table, keeping the same function-pointers as in `Base::__vtable` but replacing those slots that correspond to overrides. For instance, if `Der` overrides `virt0()` through `virt2()` and inherits the others as-is, `Der`’s v-table might look something like this (pretend `Der` doesn’t add any new virtuals):

```
// Pseudo-code (not C++, not C) for a static table defined within file Der.cpp
// Pretend FunctionPtr is a generic pointer to a generic member function
// (Remember: this is pseudo-code, not C++ code)
FunctionPtr Der::__vtable[5] = {
  &Der::virt0, &Der::virt1, &Der::virt2, &Base::virt3, &Base::virt4
                                      ↑↑↑↑          ↑↑↑↑ // Inherited as-is
};
```

In step #3, the compiler adds a similar pointer-assignment at the beginning of each of `Der`’s constructors. The idea is to change each `Der` object’s v-pointer so it points at its class’s v-table. (This is not a second v-pointer; it’s the same v-pointer that was defined in the base class, `Base`; remember, the compiler does not repeat step #2 in class `Der`.)

Finally, let’s see how the compiler implements a call to a virtual function. Your code might look like this:

```
// Your original C++ code
void mycode(Base* p)
{
  p->virt3();
}
```

The compiler has no idea whether this is going to call `Base::virt3()` or `Der::virt3()` or perhaps the `virt3()` method of another derived class that doesn’t even exist yet. It only knows for sure that you are calling `virt3()` which happens to be the function in slot #3 of the v-table. It rewrites that call into something like this:

```
// Pseudo-code that the compiler generates from your C++
void mycode(Base* p)
{
  p->__vptr[3](p);
}
```

On typical hardware, the machine-code is two 'load's plus a call:

1. The first load gets the v-pointer, storing it into a register, say r1.
2. The second load gets the word at `r1 + 3*4` (pretend function-pointers are 4-bytes long, so `r1 + 12` is the pointer to the right class's `virt3()` function). Pretend it puts that word into register r2 (or r1 for that matter).
3. The third instruction calls the code at location r2.

Conclusions:

- Objects of classes with virtual functions have only a small space-overhead compared to those that don't have virtual functions.
- Calling a virtual function is fast — almost as fast as calling a non-virtual function.
- You don't get any additional per-call overhead no matter how deep the inheritance gets. You could have 10 levels of inheritance, but there is no "chaining" — it's always the same — fetch, fetch, call.

Caveat: I've intentionally ignored multiple inheritance, virtual inheritance and RTTI. Depending on the compiler, these can make things a little more complicated. If you want to know about these things, DO NOT EMAIL ME, but instead ask comp.lang.c++.

Caveat: Everything in this FAQ is compiler-dependent. Your mileage may vary.

How can a member function in my derived class call the same function from its base class?

Use `Base::f();`

Let's start with a simple case. When you call a non-virtual function, the compiler obviously doesn't use the virtual-function mechanism. Instead it calls the function by name, using the fully qualified name of the member function. For instance, the following C++ code...

```
void mycode(Fred* p)
{
  p->goBowling();  // Pretend Fred::goBowling() is non-virtual
}
```

...might get compiled into something like this C-like code (the `p` parameter becomes the `this` object within the member function):

```
void mycode(Fred* p)
{
  __Fred__goBowling(p);  // Pseudo-code only; not real
}
```

The actual name-mangling scheme is more involved than the simple one implied above, but you get the idea. The point is that there is nothing strange about this particular case — it resolves to a normal function more-or-less like `printf()`.

Now for the case being addressed in the question above: When you call a virtual function using its fully-qualified name (the class-name followed by "`::`"), the compiler does not use the virtual call mechanism, but instead uses the same mechanism as if you called a non-virtual function. Said another way, it calls the function *by name* rather than by slot-number. So if you want code within derived class `Der` to call `Base::f()`, that is, the version of `f()` defined in its base class `Base`, you should write:

```
void Der::f()
{
  Base::f();  // Or, if you prefer, this->Base::f();
}
```

The complier will turn that into something vaguely like the following (again using an overly simplistic name-mangling scheme):

```
void __Der__f(Der* this)  // Pseudo-code only; not real
{
  __Base__f(this);        // Pseudo-code only; not real
}
```

I have a heterogeneous list of objects, and my code needs to do class-specific things to the objects. Seems like this ought to use dynamic binding but can't figure it out. What should I do?

It's surprisingly easy.

Suppose there is a base class `Vehicle` with derived classes `Car` and `Truck`. The code traverses a list of `Vehicle` objects and does different things depending on the type of `Vehicle`. For example it might weigh the `Truck` objects (to make sure they're not carrying too heavy of a load) but it might do something different with a `Car` object — check the registration, for example.

The initial solution for this, at least with most people, is to use an `if` statement. E.g., "if the object is a `Truck`, do this, else if it is a `Car`, do that, else do a third thing":

```
typedef std::vector<Vehicle*>  VehicleList;
void myCode(VehicleList& v)
{
  for (VehicleList::iterator p = v.begin(); p != v.end(); ++p) {
    Vehicle& v = **p;  // just for shorthand
    // generic code that works for any vehicle...
    // ...
    // perform the "foo-bar" operation.
    // note: the details of the "foo-bar" operation depend
    // on whether we're working with a car or a truck.
    if (v is a Car) {
      // car-specific code that does "foo-bar" on car v
      // ...
    } else if (v is a Truck) {
      // truck-specific code that does "foo-bar" on truck v
      // ...
    } else {
      // semi-generic code that does "foo-bar" on something else
      // ...
    }
    // generic code that works for any vehicle...
    // ...
  }
}
```

The problem with this is what I call “else-if-heimer’s disease” (say it fast and you’ll understand). The above code gives you else-if-heimer’s disease because eventually you’ll forget to add an `else if` when you add a new derived class, and you’ll probably have a bug that won’t be detected until run-time, or worse, when the product is in the field.

The solution is to use dynamic binding rather than dynamic typing. Instead of having (what I call) the live-code dead-data metaphor (where the code is alive and the car/truck objects are relatively dead), we move the code into the data. This is a slight variation of Bertrand Meyer’s *Law of Inversion*.

The idea is simple: use the *description* of the code within the `{...}` blocks of each `if` (in this case it is “the foo-bar operation”; obviously your name will be different). Just pick up this descriptive name and use it as the name of a new `virtual` member function in the base class (in this case we’ll add a `fooBar()` member function to class `Vehicle`).

```
class Vehicle {
public:
  // performs the "foo-bar" operation
  virtual void fooBar() = 0;
};
```

Then you remove the whole `if...else if` ... block and replace it with a simple call to this `virtual` function:

```
typedef std::vector<Vehicle*>  VehicleList;
void myCode(VehicleList& v)
{
  for (VehicleList::iterator p = v.begin(); p != v.end(); ++p) {
    Vehicle& v = **p;  // just for shorthand
    // generic code that works for any vehicle...
    // ...
    // perform the "foo-bar" operation.
    v.fooBar();
    // generic code that works for any vehicle...
    // ...
  }
}
```

Finally you *move* the code that used to be in the `{...}` block of each `if` into the `fooBar()` member function of the appropriate derived class:

```
class Car : public Vehicle {
public:
  virtual void fooBar();
};
void Car::fooBar()
{
  // car-specific code that does "foo-bar" on 'this'
  // this is the code that was in {...} of if (v is a Car)
}
class Truck : public Vehicle {
public:
  virtual void fooBar();
};
void Truck::fooBar()
{
  // truck-specific code that does "foo-bar" on 'this'
  // this is the code that was in {...} of if (v is a Truck)
}
```

If you actually have an `else` block in the original `myCode()` function (see above for the “semi-generic code that does the ‘foo-bar’ operation on something other than a Car or Truck”), change `Vehicle`’s `fooBar()` from pure virtual to plain virtual and move the code into that member function:

```
class Vehicle {
public:
  // performs the "foo-bar" operation
  virtual void fooBar();
};
void Vehicle::fooBar()
{
  // semi-generic code that does "foo-bar" on something else
  // this is the code that was in {...} of the else
  // you can think of this as "default" code...
}
```

That's it!

The point, of course, is that we try to avoid decision logic with decisions based on the kind-of derived class you're dealing with. In other words, you're trying to avoid `if the object is a car` *do xyz*, `else if it's a truck` *do pqr*, etc., because that leads to else-if-heimer's disease.

When should my destructor be `virtual`?

When someone will `delete` a derived-class object via a base-class pointer.

In particular, here's when you need to make your destructor `virtual`:

- *if* someone will derive from your class,
- *and if* someone will say `new Derived`, where `Derived` is derived from your class,
- *and if* someone will say `delete p`, where the actual object's type is `Derived` but the pointer `p`'s type is your class.

Confused? Here's a simplified rule of thumb that usually protects you and usually doesn't cost you anything: make your destructor `virtual` if your class has *any* `virtual` functions. Rationale:

- that *usually* protects you because most base classes have at least one `virtual` function.
- that *usually* doesn't cost you anything because there is no added per-object space-cost for the second or subsequent `virtual` in your class. In other words, you've already paid all the per-object space-cost that you'll ever pay once you add the first `virtual` function, so the `virtual` destructor doesn't add any additional per-object space cost. (Everything in this bullet is *theoretically* compiler-specific, but in practice it will be valid on almost all compilers.)

Note: in a derived class, if your base class has a `virtual` destructor, your own destructor is automatically `virtual`. You might need an explicitly defined destructor for other reasons, but there's no need to redeclare a destructor simply to make sure it is `virtual`. No matter whether you declare it with the `virtual` keyword, declare it without the `virtual` keyword, or don't declare it at all, it's still `virtual`.

By the way, if you're interested, here are the mechanical details of *why* you need a `virtual` destructor when someone says `delete` using a `Base` pointer that's pointing at a `Derived` object. When you say `delete p`, and the class of `p` has a `virtual` destructor, the destructor that gets invoked is the one associated with the type of the object `*p`, not necessarily the one associated with the type of the pointer. This is A Good Thing. In fact, violating that rule makes your program undefined. The technical term for that is, "Yuck."

Why are destructors not `virtual` by default?

Because many classes are not designed to be used as base classes. Virtual functions make sense only in classes meant to act as interfaces to objects of derived classes (typically allocated on a heap and accessed through pointers or references).

So when should I declare a destructor virtual? Whenever the class has at least one virtual function. Having virtual functions indicate that a class is meant to act as an interface to derived classes, and when it is, an object of a derived class may be destroyed through a pointer to the base. For example:

```
class Base {
    // ...
    virtual ~Base();
};
class Derived : public Base {
    // ...
    ~Derived();
};
void f()
{
    Base* p = new Derived;
    delete p;   // virtual destructor used to ensure that ~Derived is called
}
```

Had `Base`'s destructor not been virtual, `Derived`'s destructor would not have been called – with likely bad effects, such as resources owned by `Derived` not being freed.

What is a "`virtual` constructor"?

An idiom that allows you to do something that C++ doesn't directly support.

You can get the effect of a `virtual` constructor by a `virtual` `clone()` member function (for copy constructing), or a `virtual` `create()` member function (for the default constructor).

```
class Shape {
public:
  virtual ~Shape() { }                 // A virtual destructor
  virtual void draw() = 0;             // A pure virtual function
  virtual void move() = 0;
  // ...
  virtual Shape* clone()  const = 0;   // Uses the copy constructor
  virtual Shape* create() const = 0;   // Uses the default constructor
};
class Circle : public Shape {
public:
  Circle* clone()  const;   // Covariant Return Types; see below
  Circle* create() const;   // Covariant Return Types; see below
  // ...
};
Circle* Circle::clone()  const { return new Circle(*this); }
Circle* Circle::create() const { return new Circle();      }
```

In the `clone()` member function, the `new Circle(*this)` code calls `Circle`'s copy constructor to copy the state of `this` into the newly created `Circle` object. (Note: unless `Circle` is known to be final (AKA a leaf), you can reduce the chance of slicing by making its copy constructor `protected`.) In the `create()` member function, the `new Circle()` code calls `Circle`'s default constructor.

Users use these as if they were "`virtual` constructors":

```
void userCode(Shape& s)
{
  Shape* s2 = s.clone();
  Shape* s3 = s.create();
  // ...
  delete s2;    // You need a virtual destructor here
  delete s3;
}
```

This function will work correctly regardless of whether the `Shape` is a `Circle`, `Square`, or some other kind-of `Shape` that doesn't even exist yet.

Note: The return type of `Circle`'s `clone()` member function is intentionally different from the return type of `Shape`'s `clone()` member function. This is called *Covariant Return Types*, a feature that was not originally part of the language. If your compiler complains at the declaration of `Circle* clone() const` within class `Circle` (e.g., saying "The return type is different" or "The member function's type differs from the base class virtual function by return type alone"), you have an old compiler and you'll have to change the return type to `Shape*`.

Why don't we have `virtual` constructors?

A virtual call is a mechanism to get work done given partial information. In particular, `virtual` allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

Techniques for using an indirection when you ask to create an object are often referred to as "Virtual constructors". For example, see TC++PL3 15.6.2.

For example, here is a technique for generating an object of an appropriate type using an abstract class:

```
struct F {  // interface to object creation functions
   virtual A* make_an_A() const = 0;
   virtual B* make_a_B() const = 0;
};
void user(const F& fac)
{
   A* p = fac.make_an_A(); // make an A of the appropriate type
   B* q = fac.make_a_B();  // make a B of the appropriate type
   // ...
}
struct FX : F {
   A* make_an_A() const { return new AX(); } // AX is derived from A
   B* make_a_B() const { return new BX();  } // BX is derived from B
};
struct FY : F {
   A* make_an_A() const { return new AY(); } // AY is derived from A
   B* make_a_B() const { return new BY();  } // BY is derived from B
};
int main()
{
   FX x;
   FY y;
   user(x);    // this user makes AXs and BXs
   user(y);    // this user makes AYs and BYs
   user(FX()); // this user makes AXs and BXs
   user(FY()); // this user makes AYs and BYs
   // ...
}
```

This is a variant of what is often called "the factory pattern". The point is that `user()` is completely isolated from knowledge of classes such as `AX` and `AY`.