Inheritance — What your mother never told you

How can I set up my class so it won't be inherited from?

Just declare the class `final`.

But also ask yourself why you want to? There are two common answers:

- For efficiency: to avoid your function calls being `virtual`.
- For safety: to ensure that your class is not used as a base class (for example, to be sure that you can copy objects without fear of slicing).

In today's usual implementations, calling a virtual function entails fetching the "vptr" (i.e. the pointer to the virtual table) from the object, indexing into it via a constant, and calling the function indirectly via the pointer to function found at that location. A regular call is most often a direct call to a literal address. Although a virtual call seems to be a lot more work, the right way to judge costs is in comparison to the work actually carried by the function. If that work is significant, the cost of the call itself is negligible by comparison and often cannot be measured. If, however, the function body is simple (i.e. an accessor or a forward), the cost of a virtual call can be measurable and sometimes significant.

The virtual function call mechanism is typically used only when calling through a pointer or a reference. When calling a function directly for a named object (e.g. one allocated on the stack of the caller), the compiler inserts code for a regular call. Note, however, that frequent such use may indicate other problems with the design - virtual functions work only in tandem with polymorphism and indirect use (pointers and references). Such cases may warrant a design review for overuse of `virtual`.

How can I set up my member function so it won't be overridden in a derived class?

Just declare the function `final`.

But again, ask yourself why you want to. See the reasons under given for final classes.

Is it okay for a non-`virtual` function of the base class to call a `virtual` function?

Yes. It's sometimes (*not always!*) a great idea. For example, suppose all `Shape` objects have a common algorithm for printing, but this algorithm depends on their area and they all have a potentially different way to compute their area. In this case `Shape`'s `area()` member function would necessarily have to be `virtual` (probably pure `virtual`) but `Shape::print()` could, if we were guaranteed no derived class wanted a different algorithm for printing, be a non-`virtual` defined in the base class `Shape`.

```
#include "Shape.h"
void Shape::print() const
{
    float a = this->area();  // area() is pure virtual
    // ...
}
```

That last FAQ confuses me. Is it a different strategy from the other ways to use `virtual` functions? What's going on?

Yes, it is a different strategy. Yes, there really are two different basic ways to use `virtual` functions:

1. Suppose you have the situation described in the previous FAQ: you have a member function whose overall structure is the same for each derived class, but has little pieces that are different in each derived class. So the algorithm is the same, but the primitives are different. In this case you'd write the overall algorithm in the base class as a `public` member function (that's sometimes non-`virtual`), and you'd write the little pieces in the derived classes. The little pieces would be declared in the base class (they're often `protected`, they're often pure `virtual`, and they're *certainly* virtual), and they'd ultimately be defined in each derived class. The most critical question in this situation is whether or not the `public` member function containing the overall algorithm should be `virtual`. The answer is to make it `virtual` if you think that some derived class might need to override it.
2. Suppose you have the exact opposite situation from the previous FAQ, where you have a member function whose overall structure is different in each derived class, yet it has little pieces that are the same in most (if not all) derived classes. In this case you'd put the overall algorithm in a `public virtual` that's ultimately defined in the derived classes, and the little pieces of common code can be written once (to avoid code duplication) and stashed somewhere (anywhere!). A common place to stash the little pieces is in the `protected` part of the base class, but that's not necessary and it might not even be best. Just find a place to stash them and you'll be fine. Note that if you do stash them in the base class, you should normally make them `protected`, since normally they do things that `public` users don't need/want to do. Assuming they're `protected`, they probably shouldn't be `virtual`: if the derived class doesn't like the behavior in one of them, it doesn't have to call that member function.

For emphasis, the above list is a both/and situation, not an either/or situation. In other words, you don't have to choose between these two strategies on any given class. It's perfectly normal to have member function `f()` correspond to strategy #1 while member function `g()` corresponds to strategy #2. In other words, it's perfectly normal to have both strategies working in the same class.

Should I use protected virtuals instead of public virtuals?

Sometimes yes, sometimes no.

First, stay away from always/never rules, and instead use whichever approach is the best fit for the situation. There are at least two good reasons to use protected virtuals (see below), but just because you are sometimes better off with protected virtuals does not mean you should always use them. Consistency and symmetry are good up to a point, but at the end of the day the most important metrics are cost + schedule + risk, and unless an idea materially improves cost and/or schedule and/or risk, it's just symmetry for symmetry's sake (or consistency for consistency's sake, etc.).

The cheapest + fastest + lowest risk approach in my experience ends up resulting in most virtuals being public, with protected virtuals being used whenever you have either of these two cases: the situation discussed in the previous FAQ or the situation discussed in relation to the hiding rule.

The latter deserves some additional commentary. Pretend you have a base class with a set of overloaded virtuals. To make the example easy, pretend there are just two: `virtual void f(int)` and `virtual void f(double)`. The idea of the *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* idiom is to change the public overloaded member functions to non-virtuals, and make those call protected non-overloaded virtuals.

Code using public overloaded virtuals:

```
class Base {
public:
  virtual void f(int x);    // May or may not be pure virtual
  virtual void f(double x); // May or may not be pure virtual
};
```

Improving this via the *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* idiom:

```
class Base {
public:
  void f(int x)    { f_int(x); }  // Non-virtual
  void f(double x) { f_dbl(x); }  // Non-virtual
protected:
  virtual void f_int(int);
  virtual void f_dbl(double);
};
```

Here's an overview of the original code:

Member function	Public?	Inline?	Virtual?	Overloaded?
`f(int)` & `f(double)`	Yes	No	Yes	Yes

Here's an overview of the improved code that uses the *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* idiom:

Member function	Public?	Inline?	Virtual?	Overloaded?
`f(int)` & `f(double)`	Yes	Yes	No	Yes
`f_int(int)` & `f_dbl(double)`	No	No	Yes	No

The reason I and others use this idiom is to make life easier and less error-prone for the developers of the derived classes. Remember the goals stated above: schedule + cost + risk? Let's evaluate this Idiom in light of those goals. From a cost/schedule standpoint, the base class (singular) is slightly larger but the derived classes (plural) are slightly smaller, for a net (small) improvement in schedule and cost. The more signicant improvement is in risk: the idiom packs the complexity of properly managing the hiding rule into the base class (singular). This means the derived classes (plural) more-or-less automatically handle the hiding rule, so the various developers who produce those derived classes can remain almost completely focused on the details of the derived classes themselves — they need not concern themselves with the (subtle and often misunderstood) hiding rule. This greatly reduces the chance that the writers of the derived classes will screw up the hiding-rule.

With apologies to Mr. Spock, the needs of the many *(the derived classes (plural))* outweigh the needs of the one *(the base class (singular))*.

(Read up on the Hiding Rule for why you need to be careful about overriding some-but-not-all of a set of overloaded member functions, and therefore why the above makes life easier on derived classes.)

When should someone use private virtuals?

When you need to make specific behavior in a base class customizable in derived classes, while protecting the semantics of the interface (and/or the base algorithm therein), which is defined in public member functions that call private virtual member functions.

One case where private virtuals show up is when implementing the Template Method design pattern. Some experts, e.g., Herb Sutter's C/C++ Users Journal article Virtuality, advocate it as a best practice to always define virtual functions private, unless there is a good reason to make them protected. Virtual functions, in their view, should never be public, because they define the class' interface, which must remain consistent in all derived classes. Protected and private virtuals define the class' customizable behavior, and there is no need to make them public. A public virtual function would define both interface and a customization point, a duality that could reflect weak design.

By the way, it confuses most novice C++ programmers that private virtuals can be overridden, let alone are valid at all. We were all taught that private members in a base class are not accessible in classes derived from it, which is correct. However this inaccessibility *by* the derived class does not have anything to do with the virtual call mechanism, which is *to* the derived class. Since that might confuse novices, the C++ FAQ formerly recommended using protected virtuals rather than private virtuals. However the private virtual approach is now common enough that confusion of novices is less of a concern.

You might ask, What good is a function that the derived class can't call? Even though the derived class can't call it in the base class, the base class can call it which effectively calls down to the (appropriate) derived class. And that's what the Template Method pattern is all about.

Think of "Back to the Future." Assume the base class is written last year, and you are about to create a new derived class later today. The base class' member functions, which might have been compiled and stuck into a library months ago, will call the private (or protected) virtual, and that will effectively "call into the future" - the code which was compiled months ago will call code that doesn't even exist yet - code you are about to write in the next few minutes. You can't access private members of the base class - you can't reach into the past, but the past can reach into the future and call your member functions which you haven't even written yet.

Here is what that Template Method pattern looks like:

```
1.  class MyBaseClass {
2.  public:
3.    void myOp();
4.  private:
5.    virtual void myOp_step1() = 0;
6.    virtual void myOp_step2();
7.  };
8.  void MyBaseClass::myOp()
9.  {
10.   // Pre-processing...
11.   myOp_step1();  // call into the future - call the derived class
12.   myOp_step2();  // optionally the future - this one isn't pure virtual
13.   // Post-processing...
14. }
15. void MyBaseClass::myOp_step2()
16. {
17.   // this is "default" code - it can optionally be customized by a derived class
18. }
```

In this example, public member function `MyBaseClass::myOp()` implements the interface and basic algorithm to perform some operation. The pre- and post-processing, as well as the sequence of step 1 and step 2, are intentionally fixed and cannot be customized by a derived class. If `MyBaseClass::myOp()` was virtual, the integrity of that algorithm would be seriously compromised. Instead, customization is restricted to specific "pieces" of the algorithm, implemented in the two private virtual functions. This enforces better compliance of derived classes to the original intent embodied in the base class, and also makes customization easier - the derived class' author needs to write less code.

If `MyBaseClass::myOp_step2()` might need to be called by the derived class, for example, if the derived class might need (or want) to use that code to simplify its own code, then that can be promoted from a private virtual to a protected virtual. If that is not possible because the base class belongs to a different organization, as a band-aid the code can be copied.

(At this point I can almost read your thoughts: "What? Copy code??!? Are you KIDDING??!? That would increase maintenance cost and duplicate bugs!! Are you CRAZY??!?" Whether I'm crazy remains to be seen, but I am experienced enough to realize life sometimes paints you into a corner. If the base class can't be modified, sometimes the "least bad" of the bad alternatives is to copy some code. Remember, one size does not fit all, and "think" is not a four-letter word. So hold your nose and do whatever is the least bad thing. Then shower. Twice. But *if* you risk the team's success because you are waiting for some third party to change their base class, or if you use `#define` to change the meaning of `private`, you might have chosen a worse evil. And oh yea, if you copy the code, mark it with a big fat comment so I won't come along and think *you* are crazy!! 🙂.)

On the other hand, if you are creating the base class and if you aren't sure whether derived class's might want to call `MyBaseClass::myOp_step2()`, you can declare it protected just in case. And in that case, you'd better put a big fat comment next to it so Herb doesn't come along and think you're crazy! Either way, somebody is going to think you're crazy.

When my base class's constructor calls a `virtual` function on its `this` object, why doesn't my derived class's override of that `virtual` function get invoked?

Because that would be very dangerous, and C++ is protecting you from that danger.

The rest of this FAQ gives a rationale for why C++ needs to protect you from that danger, but before we start that, be advised that you can get the *effect as if* dynamic binding worked on the `this` object even during a constructor via The Dynamic Binding During Initialization Idiom.

You *can* call a virtual function in a constructor, but be careful. It may not do what you expect. In a constructor, the virtual call mechanism is disabled because overriding from derived classes hasn't yet happened. Objects are constructed from the base up, "base before derived".

Consider:

```
#include<string>
#include<iostream>
using namespace std;
class B {
public:
    B(const string& ss) { cout << "B constructor\n"; f(ss); }
    virtual void f(const string&) { cout << "B::f\n";}
};
class D : public B {
public:
    D(const string & ss) :B(ss) { cout << "D constructor\n";}
    void f(const string& ss) { cout << "D::f\n"; s = ss; }
private:
    string s;
};
int main()
{
    D d("Hello");
}
```

the program compiles and produce

```
B constructor
B::f
D constructor
```

Note not `D::f`. Consider what would happen if the rule were different so that `D::f()` was called from `B::B()`: Because the constructor `D::D()` hadn't yet been run, `D::f()` would try to assign its argument to an uninitialized string `s`. The result would most likely be an immediate crash. So fortunately the C++ language doesn't let this happen: it makes sure any call to `this->f()` that occurs while control is flowing through `B`'s constructor will end up invoking `B::f()`, not the override `D::f()`.

Destruction is done "derived class before base class", so virtual functions behave as in constructors: Only the local definitions are used – and no calls are made to overriding functions to avoid touching the (now destroyed) derived class part of the object.

For more details see D&E 13.2.4.2 or TC++PL3 15.4.3.

It has been suggested that this rule is an implementation artifact. It is not so. In fact, it would be noticeably easier to implement the unsafe rule of calling virtual functions from constructors exactly as from other functions. However, that would imply that no virtual function could be written to rely on invariants established by base classes. That would be a terrible mess.

Okay, but is there a way to *simulate* that behavior *as if* dynamic binding worked on the `this` object within my base class's constructor?

Yes: the *Dynamic Binding During Initialization* idiom (AKA Calling Virtuals During Initialization).

To clarify, we're talking about the situation when `Base`'s constructor calls virtual functions on its `this` object:

```
class Base {
public:
  Base();
  // ...
  virtual void foo(int n) const; // often pure virtual
  virtual double bar() const;    // often pure virtual
  // if you don't want outsiders calling these, make them protected
};
Base::Base()
{
  // ...
  foo(42);  // Warning: does NOT dynamically bind to the derived class
  bar();    // (ditto)
  // ...
}
class Derived : public Base {
public:
  // ...
  virtual void foo(int n) const;
  virtual double bar() const;
};
```

This FAQ shows some ways to *simulate* dynamic binding *as if* the calls made in `Base`'s constructor dynamically bound to the `this` object's derived class. The ways we'll show have tradeoffs, so choose the one that best fits your needs, or make up another.

The first approach is a two-phase initialization. In Phase I, someone calls the actual constructor; in Phase II, someone calls an "init" function on the object. Dynamic binding on the `this` object works fine during Phase II, and Phase II is *conceptually* part of construction, so we simply move some code from the original `Base::Base()` into `Base::init()`.

```
class Base {
public:
  void init();  // may or may not be virtual
  // ...
  virtual void foo(int n) const; // often pure virtual
  virtual double bar() const;    // often pure virtual
};
void Base::init()
{
  // Almost identical to the body of the original Base::Base()
  // ...
  foo(42);
  bar();
  // ...
}
class Derived : public Base {
public:
  // ...
  virtual void foo(int n) const;
  virtual double bar() const;
};
```

The only remaining issues are determining *where* to call Phase I and *where* to call Phase II. There are many variations on where these calls can live; we will consider two.

The first variation is simplest initially, though the code that actually wants to create objects requires a tiny bit of programmer self-discipline, which in practice means you're doomed. Seriously, if there are only one or two places that actually create objects of this hierarchy, the programmer self-discipline is quite localized and shouldn't cause problems.

In this variation, the code that is creating the object explicitly executes both phases. When executing Phase I, the code creating the object either knows the object's exact class (e.g., `new Derived()` or perhaps a local `Derived` object), or doesn't know the object's exact class (e.g., the virtual constructor idiom or some other factory). The "doesn't know" case is strongly preferred when you want to make it easy to plug-in new derived classes.

Note: Phase I often, but not always, allocates the object from the heap. When it does, you should store the pointer in some sort of managed pointer, such as a std::unique_ptr, a reference counted pointer, or some other object whose destructor deletes the allocation. This is the best way to prevent memory leaks when Phase II might throw exceptions. The following example assumes Phase I allocates the object from the heap.

```
#include <memory>
void joe_user()
{
  std::unique_ptr<Base> p( /*...somehow create a Derived object via new...*/ );
  p->init();
  // ...
}
```

The second variation is to combine the first two lines of the `joe_user` function into some `create` function. That's almost always the right thing to do when there are lots of `joe_user`-like functions. For example, if you're using some kind of factory, such as a registry and the virtual constructor idiom, you could move those two lines into a static member function called `Base::create()`:

```
#include <memory>
class Base {
public:
  // ...
  using Ptr = std::unique_ptr<Base>;  // type aliases simplify the code
  static Ptr create();
  // ...
};
Base::Ptr Base::create()
{
  Ptr p( /*...use a factory to create a Derived object via new...*/ );
  p->init();
  return p;
}
```

This simplifies all the `joe_user`-like functions (a little), but more importantly, it reduces the chance that any of them will create a `Derived` object without also calling `init()` on it.

```
void joe_user()
{
  Base::Ptr p = Base::create();
  // ...
}
```

If you're sufficiently clever and motivated, you can even *eliminate* the chance that someone could create a `Derived` object without also calling `init()` on it. An important step in achieving that goal is to make Derived's constructors, including its copy constructor, protected or private..

The next approach does not rely on a two-phase initialization, instead using a second hierarchy whose only job is to house member functions `foo()` and `bar()`. This approach doesn't always work, and in particular it doesn't work in cases when `foo()` and `bar()`

need to access the instance data declared in `Derived`, but it is conceptually quite simple and clean and is commonly used.

Let's call the base class of this second hierarchy `Helper`, and its derived classes `Helper1`, `Helper2`, etc. The first step is to move `foo()` and `bar()` into this second hierarchy:

```
class Helper {
public:
  virtual void foo(int n) const = 0;
  virtual double bar() const = 0;
};
class Helper1 : public Helper {
public:
  virtual void foo(int n) const;
  virtual double bar() const;
};
class Helper2 : public Helper {
public:
  virtual void foo(int n) const;
  virtual double bar() const;
};
```

Next, remove `init()` from `Base` (since we're no longer using the two-phase approach), remove `foo()` and `bar()` from `Base` and `Derived` (`foo()` and `bar()` are now in the `Helper` hierarchy), and change the signature of `Base`'s constructor so it takes a `Helper` by reference:

```
class Base {
public:
  Base(const Helper& h);
  // Remove init() since not using two-phase this time
  // Remove foo() and bar() since they're in Helper
};
class Derived : public Base {
public:
  // Remove foo() and bar() since they're in Helper
};
```

We then define `Base::Base(const Helper&)` so it calls `h.foo(42)` and `h.bar()` in exactly those places that `init()` used to call `this->foo(42)` and `this->bar()`:

```
Base::Base(const Helper& h)
{
  // Almost identical to the body of the original Base::Base()
  // except for the insertion of h.
  // ...
  h.foo(42);
  h.bar();
  ↑↑ // The h. occurrences are new
  // ...
}
```

Finally we change `Derived`'s constructor to pass a (perhaps temporary) object of an appropriate `Helper` derived class to `Base`'s constructor (using the init list syntax). For example, `Derived` would pass an instance of `Helper2` if it happened to contain the behaviors that `Derived` wanted for functions `foo()` and `bar()`:

```
Derived::Derived()
  : Base(Helper2())   // ← the magic happens here
{
  // ...
}
```

Note that `Derived` can pass values into the `Helper` derived class's constructor, but it *must not* pass any data members that actually live inside the `this` object. While we're at it, let's explicitly say that `Helper::foo()` and `Helper::bar()` must not access data members of the `this` object, particularly data members declared in `Derived`. (Think about when those data members are initialized and you'll see why.)

Of course the choice of which `Helper` derived class could be made out in the `joe_user`-like function, in which case it would be passed into the `Derived` ctor and then up to the `Base` ctor:

```
Derived::Derived(const Helper& h)
  : Base(h)
{
  // ...
}
```

If the `Helper` objects don't need to hold any data, that is, if each is *merely* a collection of its member functions, then you can simply pass static member functions instead. This might be simpler since it entirely eliminates the `Helper` hierarchy.

```
class Base {
public:
  using FooFn = void (*)(int);  // type aliases simplify
  using BarFn = double (*)();   //    the rest of the code
  Base(FooFn foo, BarFn bar);
  // ...
};
Base::Base(FooFn foo, BarFn bar)
{
  // Almost identical to the body of the original Base::Base()
  // except the calls are made via function pointers.
  // ...
  foo(42);
  bar();
  // ...
}
```

The `Derived` class is also easy to implement:

```
class Derived : public Base {
public:
  Derived();
  static void foo(int n); // the static is important!
  static double bar();    // the static is important!
  // ...
};
Derived::Derived()
  : Base(foo, bar)  // ← pass the function-ptrs into Base's ctor
{
  // ...
}
```

As before, the functionality for `foo()` and/or `bar()` can be passed in from the `joe_user`-like functions. In that case, `Derived`'s ctor just accepts them and passes them up into `Base`'s ctor:

```
Derived::Derived(FooFn foo, BarFn bar)
  : Base(foo, bar)
{
  // ...
}
```

A final approach is to use templates to "pass" the functionality into the derived classes. This is similar to the case where the `joe_user`-like functions choose the initializer-function or the `Helper` derived class, but instead of using function pointers or dynamic binding, it wires the code into the classes via templates.

I'm getting the same thing with destructors: calling a `virtual` on my `this` object from my base class's destructor ends up ignoring the override in the derived class; what's going on?

C++ is protecting you from yourself. What you are trying to do is very dangerous, and if the compiler did what you wanted, you'd be in worse shape.

For rationale of why C++ needs to protect you from that danger, make sure you understand what happens when a constructor calls virtuals on its this object. The situation during a destructor is analogous to that during the constructor. In particular, within the `{` body `}` of `Base::~Base()`, an object that was originally of type `Derived` has already been demoted (devolved, if you will) to an object of type `Base`. If you call a virtual function that has been overridden in class `Derived`, the call will resolve to `Base::virt()`, not to the override `Derived::virt()`. Same goes for using `typeid` on the `this` object: the `this` object really has been demoted to type `Base`; it is no longer an object of type `Derived`.

Reminder to also read this.

Should a derived class redefine ("override") a member function that is non-`virtual` in a base class?

It's legal, but it ain't moral.

Experienced C++ programmers will sometimes redefine a non-`virtual` function for efficiency (e.g., if the derived class implementation can make better use of the derived class's resources) or to get around the hiding rule. However the client-visible effects must be *identical,* since non-`virtual` functions are dispatched based on the static type of the pointer/reference rather than the dynamic type of the pointed-to/referenced object.

What's the meaning of, `Warning: Derived::f(char) hides Base::f(double)`?

It means you're going to die.

Here's the mess you're in: if `Base` declares a member function `f(double x)`, and `Derived` declares a member function `f(char c)` (same name but different parameter types and/or constness), then the `Base` `f(double x)` is "hidden" rather than "overloaded" or "overridden" (even if the Base `f(double x)` is virtual).

```
class Base {
public:
  void f(double x);  // Doesn't matter whether or not this is virtual
};
class Derived : public Base {
public:
  void f(char c);  // Doesn't matter whether or not this is virtual
};
int main()
{
  Derived* d = new Derived();
  Base* b = d;
  b->f(65.3);  // Okay: passes 65.3 to f(double x)
  d->f(65.3);  // Bizarre: converts 65.3 to a char ('A' if ASCII) and passes it to f(char c); does NOT call f(double x)!!
  delete d;
  return 0;
}
```

Here's how you get out of the mess: `Derived` must have a `using` declaration of the hidden member function. For example,

```
class Base {
public:
  void f(double x);
};
class Derived : public Base {
public:
  using Base::f;  // This un-hides Base::f(double x)
  void f(char c);
};
```

If the `using` syntax isn't supported by your compiler, redefine the hidden `Base` member function(s), even if they are non-virtual. Normally this re-definition merely calls the hidden `Base` member function using the `::` syntax. E.g.,

```
class Derived : public Base {
public:
  void f(double x) { Base::f(x); }  // The redefinition merely calls Base::f(double x)
  void f(char c);
};
```

Note: the hiding problem also occurs if class `Base` declares a member function `f(char)`.

Note: warnings are not part of the standard, so your compiler may or may not give the above warning.

Note: nothing gets hidden when you have a base-pointer. Think about it: what a derived class does or does not do is irrelevant when the compiler is dealing with a base-pointer. The compiler might not even know that the particular derived class exists. Even if it knows of the existence of some particular derived class, it cannot assume that a specific base-pointer necessarily points at an object of that particular derived class. Hiding takes place when you have a derived pointer, not when you have a base pointer.

Why doesn't overloading work for derived classes?

That question (in many variations) are usually prompted by an example like this:

```
#include<iostream>
using namespace std;
class B {
public:
    int f(int i) { cout << "f(int): "; return i+1; }
    // ...
};
class D : public B {
public:
    double f(double d) { cout << "f(double): "; return d+1.3; }
    // ...
};
int main()
{
    D* pd = new D;
    cout << pd->f(2) << '\n';
    cout << pd->f(2.3) << '\n';
    delete pd;
}
```

which will produce:

```
f(double): 3.3
f(double): 3.6
```

rather than the

```
f(int): 3
f(double): 3.6
```

that some people (wrongly) guessed.

In other words, there is no overload resolution *between* `D` and `B`. Overload resolution conceptually happens in one scope at a time: The compiler looks into the scope of `D`, finds the single function `double f(double)`, and calls it. Because it found a match, it never

bothers looking further into the (enclosing) scope of `B`. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule. (See D&E or TC++PL4 for details).

But what if I want to create an overload set of all my `f()` functions from my base and derived class? That's easily done using a `using` -declaration, which asks to bring the functions into the scope:

```
class D : public B {
public:
    using B::f; // make every f from B available
    double f(double d) { cout << "f(double): "; return d+1.3; }
    // ...
};
```

Given that modification, the output will be:

```
f(int): 3
f(double): 3.6
```

That is, overload resolution was applied to `B`'s `f()` and `D`'s `f()` to select the most appropriate `f()` to call.

What does it mean that the "virtual table" is an unresolved external?

If you get a link error of the form "`Error: Unresolved or undefined symbols detected: virtual table for class Fred`," you probably have an undefined `virtual` member function in `class Fred`.

The compiler typically creates a magical data structure called the "virtual table" for classes that have `virtual` functions (this is how it handles dynamic binding). Normally you don't have to know about it at all. But if you forget to define a `virtual` function for class `Fred`, you will sometimes get this linker error.

Here's the nitty gritty: Many compilers put this magical "virtual table" in the compilation unit that defines the first non-`inline` `virtual` function in the class. Thus if the first non-`inline` `virtual` function in `Fred` is `wilma()`, the compiler will put `Fred`'s virtual table in the same compilation unit where it sees `Fred::wilma()`. Unfortunately if you accidentally forget to define `Fred::wilma()`, rather than getting a `Fred::wilma()` is undefined, you may get a "`Fred`'s virtual table is undefined". Sad but true.