Virtuality

Virtuality

*This article appeared in **C/C++ Users Journal, 19(9), September 2001**.*

This month, I want to present up-to-date answers to two recurring questions about virtual functions. These answers then lead directly to four class design guidelines.

The questions are old, but people still keep asking them, and some of the answers have changed over time as we've gained experience with modern C++.

Virtual Question #1: Publicity vs. Privacy?

The first of the two classic questions we'll consider is this: "When should virtual functions be public, protected, or private?" The short answer is: Rarely if ever, sometimes, and by default, respectively - the same answer we've already learned for other kinds of class members.

Most of us have learned through bitter experience to make all class members private by default unless we really need to expose them. That's just good encapsulation. Certainly we've long ago learned that data members should always be private (except only in the case of C-style data structs, which are merely convenient groupings of data and are not intended to encapsulate anything). The same also goes for member functions, and so I propose the following guidelines which could be summarized as a statement about the benefits of privatization.

Guideline #1: Prefer to make interfaces nonvirtual, using Template Method.

Interestingly, the C++ standard library already overwhelmingly follows this guideline. Not counting destructors (which are discussed separately later on under Guideline #4), and not double-counting the same virtual function twice when it appears again in a specialization of a class template, here's what the standard library has:

- 6 public virtual functions, all of which are std::exception::what() and its overrides
- 142 nonpublic virtual functions

Why is this such a good idea? Let's investigate.

Traditionally, many programmers were used to writing base classes using public virtual functions to directly and simultaneously specify both the interface and the customizable behavior. For example, we might write:

```
// Example 1: A traditional base class.
//
class Widget
{
public:
  // Each of these functions might optionally be
  // pure virtual, and if so might or might not have
  // an implementation in Widget; see Item 27 in [1].
  //
  virtual int Process( Gadget& );
  virtual bool IsDone();
  // ...
};
```

The problem is that "simultaneously" part, because each virtual function is doing two jobs: It's *specifying interface* because it's public and therefore directly part of the interface Widget presents to the rest of the

world; and it's *specifying implementation detail*, namely the internally customizable behavior, because it's virtual and therefore provides a hook for derived classes to replace the base implementation of that function (if any). That a public virtual function inherently has two significantly different jobs is a sign that it's not separating concerns well and that we should consider a different approach.

What if we want to separate the specification of interface from the specification of the implementation's customizable behavior? Then we end up with something that should remind us strongly of the Template Method pattern[2], because that's exactly what it is: *[Later note: Actually it's a more restricted idiom with a form similar to that of Template Method. This idiom deserves its own name, and since writing this article I've switched to calling the idiom the Non-Virtual Interface Idiom, or NVI for short. -hps]*

```
// Example 2: A more modern base class, using
// Template Method to separate interface from
// internals.
//
class Widget
{
public:
  // Stable, nonvirtual interface.
  //
  int Process( Gadget& ); // uses DoProcess...()
  bool IsDone(); // uses DoIsDone()
  // ...
private:
  // Customization is an implementation detail that may
  // or may not directly correspond to the interface.
  // Each of these functions might optionally be
  // pure virtual, and if so might or might not have
  // an implementation in Widget; see Item 27 in [1].
  //
  virtual int DoProcessPhase1( Gadget& );
  virtual int DoProcessPhase2( Gadget& );
  virtual bool DoIsDone();
  // ...
};
```

Prefer to use Template Method to make the interface stable and nonvirtual, while delegating customizable work to nonpublic virtual functions that are responsible for implementing the customizable behavior. After all, virtual functions are designed to let derived classes customize behavior; it's better to not let publicly derived classes also customize the inherited interface, which is supposed to be consistent.

The Template Method approach has several benefits and no significant drawbacks.

First, note that the base class is now in complete control of its interface and policy, and can enforce interface preconditions and postconditions, insert instrumentation, and do any similar work all in a single convenient reusable place - the nonvirtual interface function. This promotes good class design because it lets the base class enforce the substitutability compliance of derived classes in accord with the Liskov Substitution Principle[3], to whatever extent enforcement makes sense. If efficiency is an issue, the base class can elect to check certain kinds of pre- and postconditions only in a debug mode, for example via a non-debug "release" build that completely removes the checking code from the executable image, or via a configurable debug mode that suppresses selected checking code at runtime.

Second, when we've better separated interface and implementation, we're free to make each take the form it naturally wants to take instead of trying to find a compromise that forces them to look the same. For example, notice that in Example 2 we've incidentally decided that it makes more sense for our users to see a single Process() function while allowing more flexible customization in two parts, DoProcessPhase1() and DoProcessPhase2(). And it was easy. We couldn't have done this with the public virtual version without making the separation also visible in the interface, thereby adding complexity for the user who would then have to know to call two functions in the right way. (For more discussion of a related example, see also Item 23 in *Exceptional C++*[4].)

Third, the base class is now less fragile in the face of change. We are free to change our minds later and add pre- and postcondition checking, or separate processing into more steps, or refactor, or implement a fuller interface/implementation separation using the Pimpl idiom[4], or make other modifications to Widget's customizability, without affecting the code that uses Widget. For example, it's much more difficult to start

with a public virtual function and later try to wrap it for pre- and postcondition checking after the fact, than it is to provide a dumb passthrough nonvirtual wrapper up front (even if no checking or other extra work is immediately needed) and insert the checking later. (For more discussion of how a class like Widget is less fragile and more amenable to future revision and refactoring, see the article "Virtually Yours"[5].)

"But but but," some have objected, "let's say that all the public nonvirtual function does initially is pass through to the private virtual one. It's just one stupid little line. Isn't that pretty useless, and indeed haven't we lost something? Haven't we lost some efficiency (the extra function call) and added some complexity (the extra function)?" No, and no. First, a word about efficiency: No, none is lost in practice because if the public function is a one-line passthrough declared inline, all compilers I know of will optimize it away entirely, leaving no overhead. (Indeed, some compilers will always make such a function inline and eliminate it, whether you personally really wanted it to or not, but that's another story.) Second, a word about complexity: The only complexity is the extra time it takes to write the one-line wrapper function, which is trivial. Period. That's it. *C'est tout.* The interfaces are unaffected: The class still has exactly the same number of public functions for a public user to learn, and it has exactly the same number of virtual functions for a derived class programmer to learn. Neither the interface presented to the outside world, nor the inheritance interface presented to derived classes, has become any more complex in itself for either audience. The two interfaces are just explicitly separated, is all, and that is a Good Thing.

Well, that justifies nonvirtual interfaces and tells us that virtual functions benefit from being nonpublic, but we haven't really answered whether virtual functions should be private or protected. So let's answer that:

Guideline #2: Prefer to make virtual functions private.

That's easy. This lets the derived classes override the function to customize the behavior as needed, without further exposing the virtual functions directly by making them callable by derived classes (as would be possible if the functions were just protected). The point is that virtual functions exist to allow customization; unless they also need to be invoked directly from within derived classes' code, there's no need to ever make them anything but private. But sometimes we do need to invoke the base versions of virtual functions (see the article "Virtually Yours"[5] for an example), and in that case only it makes sense to make those virtual functions protected, thus:

Guideline #3: Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.

The bottom line is that Template Method as applied to virtual functions nicely helps us to separate interface from implementation. It's possible to make the separation even more complete, of course, by completely divorcing interface from implementation using patterns like Bridge[2], idioms like Pimpl (principally for managing compile-time dependencies and exception safety guarantees)[1] [4] or the more general handle/body or envelope/letter[6], or other approaches. Unless you need a more complete interface/implementation separation, though, Template Method will often be *sufficient* for your needs. On the flip side, I am arguing that this use of Template Method is also a good idea to adopt by default and view as a *necessary* minimum separation in practice in new code. After all, it costs nothing (beyond writing an extra line of code) and buys quite a bit of pain reduction down the road.

Former communist countries are learning the benefits of privatization, in those cases where privatization makes sense. The lesson of healthy privatization is likewise not lost on good class designers. For more examples of using the Template Method pattern to privatize virtual behavior, see "Virtually Yours".[5]

Speaking of that article, did you notice that the code there presented a public virtual destructor? This brings us to the second topic of this month's column:

Virtual Question #2: What About Base Class Destructors?

The second classic question we'll consider is that old destructor chestnut: "Should base class destructors be virtual?"

Sigh. I wish this were only a frequently asked question. Alas, it's more often a frequently debated question. If I had a penny for every time I've seen this debate, I could buy a cup of coffee. Not just any old coffee, mind you - I could buy a genuine Starbucks Venti double-Valencia latte (my current favorite). Maybe even two of them, if I was willing to throw in a dime of my own.

The usual answer to this question is: "Huh? Of course base class destructors should always be virtual!" This answer is wrong, and the C++ standard library itself contains counterexamples refuting it, but it's right often enough to give the illusion of correctness.

The slightly less usual and somewhat more correct answer is: "Huh? Of course base class destructors should be virtual if you're going to delete polymorphically (i.e., delete via a pointer to base)!" This answer is technically right but doesn't go far enough.

I've recently come to conclude that the fully correct answer is this:

Guideline #4: A base class destructor should be either public and virtual, or protected and nonvirtual.

Let's see why this is so.

First, an obvious statement: Clearly any operation that will be performed through the base class interface, and that should behave virtually, should be virtual. That's true even with Template Method, above, because although the public interface function is nonvirtual, the work is delegated to a nonpublic virtual function and we get the virtual behavior that we need.

If deletion, therefore, can be performed polymorphically through the base class interface, then it must behave virtually and must be virtual. Indeed, the language requires it - if you delete polymorphically without a virtual destructor, you summon the dreaded specter of "undefined behavior," a specter I personally would rather not meet in even a moderately well-lit alley, thank you very much. Hence:

```
// Example 3: Obvious need for virtual destructor.
//
class Base { /*...*/ };
class Derived : public Base { /*...*/ };
Base* b = new Derived;
delete b; // Base::~Base() had better be virtual!
```

Note that the destructor is the one case where the Template Method pattern cannot be applied to a virtual function. Why not? Because once execution reaches the body of a base class destructor, any derived object parts have already been destroyed and no longer exist. If the Base destructor body were to call a virtual function, the virtual dispatch would reach no further down the inheritance hierarchy than Base itself. In a destructor (or constructor) body, further-derived classes just don't exist any more (or yet).

But base classes need not always allow polymorphic deletion. For example, in the standard library itself,[7] consider class templates such as std::unary_function and std::binary_function. Those two class templates look like this:

```
template <class Arg, class Result>
struct unary_function
{
  typedef Arg    argument_type;
  typedef Result result_type;
};
template <class Arg1, class Arg2, class Result>
struct binary_function
{
  typedef Arg1   first_argument_type;
  typedef Arg2   second_argument_type;
  typedef Result result_type;
};
```

Both of these templates are specifically intended to be instantiated as base classes (in order to inject those standardized typedef names into derived classes) and yet do not provide virtual destructors because they are not intended to be used for polymorphic deletion. That is, code like the following is not merely unsanctioned but downright illegal, and it's reasonable for you to assume that such code will never exist:

```
// Example 4: Illegal code that you can assume
// will never exist.
//
void f( std::unary_function* f )
{
  delete f; // error, illegal
}
```

Note that the standard tut-tuts and declares Example 4 to fall squarely into the Undefined Behavior Pit, but the standard doesn't actually require a compiler to prevent you or anyone else from writing that code (more's

the pity). It would be easy and nice - and it wouldn't break any standards-conforming C++ programs that exist today - to give std::unary_function (and other classes like it) an empty but protected destructor, in which case a compiler would actually be required to diagnose the error and toss it back in the offender's face. Maybe we'll see such a change in a future revision to the standard, maybe we won't, but it would be nice to make compilers reject such code instead of just making tut-tut noises in standardish legalese.

Finally, what if a base class is concrete (can be instantiated on its own) but also wants to support polymorphic destruction? Doesn't it need a public destructor then, since otherwise you can't easily create objects of that type? That's possible, but only if you've already violated another guideline, to wit: Don't derive from concrete classes. Or, as Scott Meyers puts it in Item 33 of *More Effective C++*,[8] "Make non-leaf classes abstract." (Admittedly, it can happen in practice - in code written by someone else, of course, not by you! - and in this one case you may have to have a public virtual destructor just to accommodate what's already a poor design. Better to refactor and fix the design, though, if you can.)

In brief, then, you're left with one of two situations. Either: a) you want to allow polymorphic deletion through a base pointer, in which case the destructor must be virtual and public; or b) you don't, in which case the destructor should be nonvirtual and protected, the latter to prevent the unwanted usage.

Summary

In summary, prefer to make base class virtual functions private (or protected if you really must). This separates the concerns of interface and implementation, which stabilizes interfaces and makes implementation decisions easier to change and refactor later. For normal base class functions:

- Guideline #1: Prefer to make interfaces nonvirtual, using Template Method.
- Guideline #2: Prefer to make virtual functions private.
- Guideline #3: Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.

For the special case of the destructor only:

- Guideline #4: A base class destructor should be either public and virtual, or protected and nonvirtual.

True, the standard library itself does not always follow these design criteria. In part, that's a reflection of how we as a community have learned over the years.

Notes

1. H. Sutter. ***More Exceptional C++*** (Addison-Wesley, 2002).

2. Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

3. B. Liskov. "Data Abstraction and Hierarchy" (*SIGPLAN Notices*, 23(5), May 1988).

4. H. Sutter. ***Exceptional C++*** (Addison-Wesley, 2000).

5. J. Hyslop and H. Sutter. **"Virtually Yours"** (*C/C++ Users Journal Experts Forum*, 18(12), December 2000).

6. J. Coplien. Advanced C++ Programming Styles and Idioms (Addison-Wesley, 1992).

7. ISO/IEC 14882:1998(E), Programming Languages - C++ (ISO and ANSI C++ standard).

8. S. Meyers. More Effective C++ (Addison-Wesley, 1996).