

This is a pre-publication draft of the column I wrote for the [January 1995](#) issue of the *C++ Report*. “Pre-publication” means this is what I sent to the *Report*, but it may not be exactly the same as what appeared in print, because the *Report* and I often make small changes after I submit the “final” draft for a column. Comments? Feel free to send me mail: smeyers@aristeia.com.

min, max, and more

For those of you who got here late, I’ve spent the last few columns explaining why it’s difficult to write assignment operators that behave the way you want them to, and I’ve used this fact to justify my decidedly radical philosophy about deriving concrete classes from other concrete classes. (In a word, *don’t*.) I have a bit more to say on this topic, but before I do, I want to embark on an unrelated, but still worthwhile subject: that of writing functions like `min` and `max`.

Actually, the digression is less unrelated than it seems, because assignment operators and `min/max` functions have something quite important in common, namely, they’re both much more difficult to implement than they appear. For purposes of the following discussion, we’ll only consider `max`, but where there’s `max`, you can be sure `min` isn’t far behind. (Think about that for a minute...)

Let’s start at the beginning. In the beginning, there was C. And Stroustrup looked down upon the C, and he spaketh, “Eh, not bad, but we can do better. Let there be C++.” And so Stroustrup created C++, and it was good. By implication, C was bad, or at least it wasn’t as good. Still, it’s important to get in touch with our roots, so let’s review how you write run-of-the-mill `max` functions in C.

Basically, you don’t. You don’t write `max functions`, you write `max macros`. If you’re just starting out, you do it like this,

```
#define max(a,b) a > b ? a : b
```

but it usually doesn’t take too many overnight debugging sessions (or merciful C programmers looking over your shoulder) to rewire your neurons so you’re incapable of writing anything but this:

```
#define min(a,b) ((a) > (b) ? (a) : (b))
```

From a C++ point of view, however, this is still pretty easy to pick apart. In *Effective C++* [cite *Effective C++*], for example, I say this about it:

This little number has so many drawbacks, it’s painful just to think about them. You’re better off playing in the freeway during rush hour.

Whenever you write a macro like this, you have to remember to parenthesize all your arguments when you write the macro body; otherwise you can run into trouble when somebody calls the macro with an expression. But even if you get that right,

look at all the weird things that can happen:

```
int a = 1, b = 0;

max(a++, b);           // a is incremented twice
max(a++, b+10);       // a is incremented once
max(a, "Hello");      // comparing ints and ptrs
```

Here you have a situation in which what happens to `a` inside `max` depends on what `a` is being compared with, plus you have lost all vestiges of type safety. In fairness, the last call will elicit a warning from many compilers, but of course the warning message says nothing about `max`, because `max` doesn't exist as far as compilers are concerned.

Fortunately, you don't need to put up with this kind of nonsense. You can get all the efficiency of a macro plus all the predictable behavior and type-safety of a regular function by using an inline function:

```
inline int max(int a, int b)
{ return a > b ? a : b; }
```

Now this isn't quite the same as the macro above, because this version of `max` can only be called with ints, but templates fix that problem quite nicely:

```
template<class T>
inline T& max(T& a, T& b)
{ return a > b ? a : b; }
```

This template generates a whole family of functions, each of which takes two objects of the same type and returns a reference to the greater of the two objects. Because you don't know what the type `T` will be, you pass and return by reference for efficiency.

That about sums up the `max` situation as far as most C++ programmers are concerned. Certainly I thought that closed the subject when I wrote those words in 1991. But, you know how things go — old and wiser, that kind of thing. So now I feel like the foregoing discussion, far from being the final word on the matter, is in fact only the end of the *prelude*.

A good place to begin the body of the discussion is with a consideration of `const` correctness. The template above is immediately suspicious, because the function parameters `a` and `b` are non-`const`, even though it makes no sense to modify `a` or `b` in the process of determining which is greater.

Now, `const` is a wonderful thing, and we want to use it whenever we can, so we might try rewriting the template like this:

```
template<class T>
inline T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

This seems like a good idea, and it will probably even sail through your compiler with no complaints — until you try to actually use it. Compilers, you see, generally don't pay too much attention to template source code until the templates are *instantiated*, at which point compilers apply the same stringent type-checking rules to the instantiated code as they do to any other C++ source code submitted to them. (It is, in fact, an interesting exercise to see just how many

syntax errors you can put in a template and still have the compiler pass over the (uninstantiated) template without complaining.) In this case, when the compiler instantiates the `max` template, it will notice that the return value of the function is a non-`const` reference to either `a` or `b`, and it will further notice that both `a` and `b` are declared `const`, so it will refuse to return a non-`const` reference to them.

It's easy enough to pacify the beast, of course. If two occurrences of `const` are good, three must be better:

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

This will compile cleanly, even when instantiated. That's good. However, it also modifies the semantics of the function slightly, and that may not be so good.

It is of great relevance here that the `max` functions (`max` is a template, so there are many such functions) return a *reference*, not a value. It is possible to justify this decision on efficiency grounds (use of a reference avoids the need to call a constructor — and later a destructor — on a temporary object returned by the functions), but that is not the primary reason why I use a reference. My justification is more philosophical. Call me a renegade if you must, but I believe that if you ask me for the maximum of an object `a` and an object `b`, the answer should be either `a` or `b`. Not a *copy* of `a` or a *copy* of `b`, but `a` or `b` itself — the actual object. That's why I return a reference.

The fact that `max` returns a reference means `max` can be used as the target of an assignment. (To be precise, it means that calls to `max` can be used as lvalues). The original template, therefore, can be used like this:

```
template<class T>
inline T& max(T& a, T& b)
{ return a > b ? a : b; }

double d1, d2;

...

max(d1, d2) = 0;    // set max of d1 and d2 to 0
```

Some people find this kind of assign-to-the-return-value-of-a-function usage deplorable, but I don't happen to be one of them. This usage seems perfectly reasonable to me. It does not seem perfectly reasonable to the compiler, however, when `max` is changed to return a `const` reference, because you can't make an assignment to a `const` object. The bottom line, then, is that if you are going to have `max` return a reference (as I believe is correct) and if you want to declare `max`'s formal parameters `const`, you must also declare `max`'s return type `const`.

There is another problem with having `max` return one of its arguments by `const` reference, one I recently discovered while perusing Marshall Cline's and Greg Lomow's new book, *C++ FAQs* [cite it here].

Suppose you decide, as I did in *Effective C++*, to omit the `const`s. You can then use `max` as an lvalue, but you are left with a different problem: you can't pass `const` arguments to `max`:

```

template<class T>
inline T& max(T& a, T& b)
{ return a > b ? a : b; }

void f(const BigNumber& n1, const BigNumber& n2)
{
    BigNumber n3 = max(n1, n2);          // error!
    ...
}

```

In `f`, you've asked the compiler to prevent you from modifying `n1` and `n2`, but `max` makes no such pledge about how it treats its arguments, so the compiler rejects the call to `max`. After all, that's its job. Still, it would certainly be convenient to be able to determine the maximum of two `BigNumber` objects, *n'est-ce pas?*

So, once again applying the precept that more is better, you decide to provide *two* `max` templates:

```

// non-const version
template<class T>
inline T& max(T& a, T& b)
{ return a > b ? a : b; }

// const version
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }

void f(const BigNumber& n1, const BigNumber& n2)
{
    BigNumber n3 = max(n1, n2);          // okay
    ...

    double d1, d2;
    ...
    max(d1, d2) = 0;                     // also okay
    ...
}

```

This is fine, as far as it goes, but it doesn't go far enough. What if you want to find the maximum of two objects, but one is `const` and one is not? For example:

```

void g(const BigNumber& n1)
{
    BigNumber n2 = 22;
    BigNumber n3 = max(n1, n2);          // call which max?
    ...
}

```

Which `max` should be instantiated and called? It clearly can't be the non-const one, because it wouldn't be legal to pass the `const` object by non-const reference. It must therefore be the `const` one, but the rules for template instantiation in the ARM [cite the ARM] require that the type of the template parameter `T` *exactly match* the type of the actual instantiation argument(s), and there is no way that `T` can simultaneously bind to both `const BigNumber` and `BigNumber`.

Adherents to the "when in doubt, get a bigger hammer" philosophy may point out that, thanks to recent modifications to the emerging ANSI/ISO standard [cite Josée Lajoie's column], you can force the issue thusly,

This isn't exactly heavenly, but it's certainly preferable to having to specify which function to call at each call site (an idea that defeats the whole purpose of overloading in the first place).

Now that I've broached the subject of implicit type conversions in the context of templates, it's worth making a brief aside to explore a simple-looking question. Namely, for what types `T` will `max` template instantiations successfully compile? To answer that question, it helps to review the body of the `max` template (and it's worth noting that the body is the same regardless of whether the parameters and return value are `const`). The body looks like this:

```
{ return a > b ? a : b; }
```

This is certainly simple enough, so it seems clear that the answer to the question is that `T` must support an `operator>` function. That's true, but it's important to be precise. More specifically, it seems clear that there must be a member function or global function called `operator>` that takes arguments of type `T`. That, however, is *not* true.

As Rob Murray points out in his book, *C++ Strategies and Tactics* [cite it here], the constraint on `T` is that the expression "`a>b`" *must be legal*. There are two ways it can be legal. First, there might be an `operator>` such as we've discussed that takes two arguments of type `T`. An equally viable alternative, however, is that an `operator>` exists for some type `T'` such that objects of type `T` can be *implicitly converted* into objects of type `T'`. For example, given a class for representing strings that can be implicitly converted into their equivalent `char*` representations,

```
class String {
public:
    ...
    operator const char*() const;
};
```

you might be surprised (and dismayed) to discover that you can compute the `max` of two `String` objects, even if there are no versions of `operator>` that take `String` objects. That's because `String` objects can be implicitly converted into `const char*` pointers, and `operator>` for such pointers is built into the language. Usually, you won't want such implicit conversions to be performed, and perhaps that's why the draft standard for the (library) `string` class provides a function called `c_str` instead of the implicit type conversion function shown above [cite Plauger's book here].

At this point, let's take stock of the things we'd like to see in a `max` function (or set of functions):

1. Offers function call semantics (including type checking), not macro semantics.
2. Supports both `const` and `non-const` arguments (including mixing the two in a single call).
3. Supports arguments of different types (where that makes sense).
4. Requires no explicit instantiation.

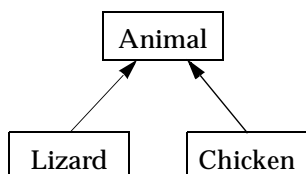
The C approach utterly fails to provide point 1, but it comes through with flying colors on points 2, 3, and 4. The template approach, on the other hand, achieves point 1 spectacularly, musters partial support for point 2, and fails miserably in the realm of point 3 unless you give up point 4.

What, then, is the best way — the *correct* way — to implement `max`? In the immortal words of Tevye, “I’ll tell you: I don’t know.” Faced with the above analysis, I increasingly find myself telling people that the macro approach may well be best, and I *hate* macros. If you know of a superior way to implement `max`, please let me know.

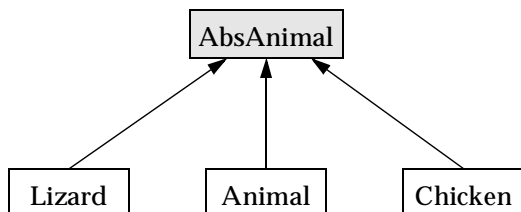
(The gall of it all is that we’re talking about the *max* function here! How can such a conceptually simple function cause so much trouble? Of course, one can say the same thing about the lowly assignment operator, and that’s how we got into this discussion in the first place.)

Well, then, enough of `mins` and `maxs`. Let us move on to other things. Pleasant things. Useful things. Things like guidelines for determining when to create abstract classes in an inheritance hierarchy.

Those of you who’ve been steadfastly sticking with me to these past few months will recall my suggestion that, when faced with a desire to have a new concrete class inherit from an existing concrete class, one should refocus that desire (design-level sublimation, as it were) and instead create a new abstract class from which both concrete classes inherit. As I put it in my July-August column, don’t do this:



Do this instead:



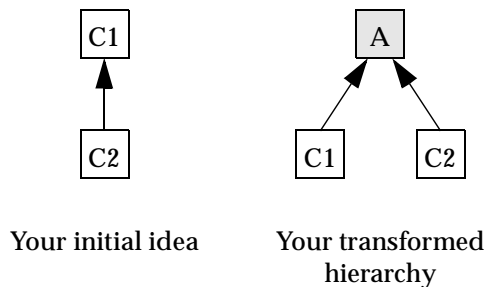
Here, shading indicates abstract classes (e.g., `AbsAnimal`) and non-shading (e.g., `Animal`) indicates concrete classes.

Of course, you can’t do this unless you have write access to the files making up the class hierarchy, but for the purposes of this discussion, let’s assume you do. In my November column, I briefly considered your options if you do not.

When I first introduced this transformation, my motivation was to come up with a way to allow you to write assignment operators that both behave correctly and can’t be invoked in such a way as to perform inadvertent partial assignments. The transformation has an equally important design-level purpose however, because it forces you to *explicitly recognize the existence of useful abstractions*. That is, it

makes you create new abstract classes for useful abstract concepts, even if you aren't initially aware of the fact that the useful concepts exist.

If that sounds a little vague, nebulous, and basically hand-wavy, that's because we're poking at the edges of object-oriented design, and OOD (as the chicest of the chic like to call it) too often has a peculiarly touchy-feely character to both its practice and its vocabulary. This notwithstanding, there *can* be a there there, and I'll show it to you in a moment. (Trust me here, I am *not* wearing a tie.) Before I do, however, it's important to identify what transformation I'm talking about. It is this: if you have two concrete classes C1 and C2, and you'd like C2 to publicly inherit from C1, you should transform that two-class hierarchy into a three-class hierarchy by creating a new abstract class A and having both C1 and C2 publicly inherit from it:



The important thing about this transformation is that it *forces* you to identify the abstract class A. Clearly C1 and C2 have something in common (that's why they're related by public inheritance), and with this transformation, you *must* identify what that something is. Furthermore, you must *formalize* the something as a class in C++, at which point it becomes more than just a vague something, it achieves the status of a formal *abstraction*, one with well-defined member functions and well-defined semantics.

The reason why it's so important to express this abstraction in the form of a class is that the abstraction *is* in your design, regardless of whether you are aware of it. If you don't perform the transformation, however, the abstraction remains hidden and muddled, because it's sharing the same class as C1. Conceptually, C1 represents concrete objects, but the abstraction does not. As long as the concept of concrete C1 objects and the abstraction represented by A remain intertwined in the single class C1, you'll have difficulty using the abstraction. Often, you will be unaware of the fact that it even exists. Only by forcing yourself to break it out explicitly can you identify exactly what the abstraction is and how it behaves. (This intertwining of abstraction and object-generator in a single class is in many ways akin to the intertwining of interface inheritance and implementation inheritance in C++'s public inheritance [cite *Effective C++* here].)

All of which leads to some dangerous thinking. After all, you may say, *every* class represents *some* kind of abstraction, so shouldn't we create two classes for every concept in our hierarchy, one being abstract (to embody the abstract part of the abstraction) and one being

concrete (to embody the object-generation part of the abstraction)? To which I say *no*, you should not. If you do, you end up with a hierarchy with too many classes. Such a hierarchy is difficult to understand, hard to maintain, and expensive to compile. That is *not* the goal of this game.

The goal is to identify *useful* abstractions and to force them — and only them — into existence as abstract classes. But how do you identify useful abstractions? Who knows what abstractions might prove useful in the future? Who can predict who's going to want to inherit from what?

Well, I don't know of a perfect way to write a `max` function, and I don't know how to predict the future uses of an inheritance hierarchy, either, but I do know one thing: the need for an abstraction in *one* context is often coincidental, but the need for an abstraction in *more than one* context is usually meaningful. Useful abstractions, then, are those that are needed in more than one context. That is, they correspond to classes that are both useful in their own right (i.e., it is useful to have objects of that type) and that are also useful for purposes of one or more derived classes.

This is precisely why the transformation is useful: it forces the introduction of a new abstract class only in those cases where an existing concrete class is about to be used as a base class, i.e., in situations where the class is about to be (re)used in a new context. I decree such abstractions *useful*, because they have, through demonstrated need, shown themselves to be so. The first time a concept is needed, we can't necessarily justify the creation of both an abstract class (for the concept) and a concrete class (for the objects corresponding to that concept), but the second time that concept is needed, we *can* justify the creation of both the abstract and the concrete classes. The transformation I've described simply mechanizes the implications of this analysis, and in so doing it forces designers and programmers to explicitly represent those abstractions that are useful, even if the software developers are not consciously aware of the existence of the useful concepts. It also happens to make it one heck of a lot easier to bring sanity to the behavior of class assignment operators.

But enough of sermon and hot air. (I'm not wearing a tie, really I'm not.) Let me give a brief example. Suppose you're working on an application that deals with moving information between computers on a network. When the information is moved, it is broken into packets and transmitted according to some protocol in the usual fashion. One could spend days (or, if you're on a tight schedule, weeks) delving into the analysis and design of such a system, but all I want to consider here is the class or classes for representing packets; we'll assume such classes make sense for this application.

Suppose you deal with only a single kind of transfer protocol and only a single kind of packet. Perhaps you've heard that other protocols and packet types exist, but your company has never supported them, nor does it have any plans to support them in the future. Should you make an abstract class for packets (for the concept that a packet represents) as well as a concrete class for the packets you'll actually be using? If you do, you might hope to be able to add new packet types later without having to change the base class for pack-

ets; that would save you the need to recompile all packet-using applications in the future if you do indeed add new packet types. But that design will require two classes, and right now you need only one (for the particular type of packets you use). Is it worth complicating your design now to allow for future extension that may never occur?

There is no single correct choice to be made here, but experience has shown that it is very difficult to design classes for concepts that we do not understand extremely well. If you were to decide to create an abstract class for packets, how likely is it that you could get it right, especially in view of the fact that your company's experience is limited to only a single packet type? Remember that you gain the benefit of an abstract class for packets only if you can design that class such that future classes can inherit from it without requiring that it be changed in any way. (If it needs to be changed, you have to recompile all packet clients, anyway, in which case you haven't really gained anything.) As Martin Carroll has argued [cite his article here], designing such classes is very difficult; it is unlikely that you could do it for a general abstract packet class unless you were extremely well versed in many different kinds of packets and in the varied contexts in which they may be used. My suggestion, then, would be to forego the declaration of an abstract class for packets, adding such a class only if you later find a need to inherit from the concrete `Packet` class.

It is important to recognize that the transformation I've described here is a way to identify the need for abstract classes, not *the* way. There are many other ways to identify good candidates for abstract classes; books on OOA and OOD are filled with them. I'm not arguing that the only time you should introduce abstract classes into a design is when you find yourself wanting to have a concrete class inherit from another concrete class. I do believe, however, that the desire to relate two concrete classes by public inheritance is indicative of a need for a new abstract class. Even if you don't buy this design argument, however, don't forget that following the transformation I've described also simplifies the implementation of our friend, the assignment operator.

Acknowledgment

Steve Clamage's insightful comments on an earlier draft of this column helped improve the version appearing here.

References:

Meyers, S. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, MA 1992.

Ellis, M. and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA 1990.

"Templates: New and improved, like your favorite detergent," Josée Lajoie, *The C++ Report*, May 1994, pp. 62-69.

C++ Strategies and Tactics, Robert Murray, Addison-Wesley, 1993, pp. 190-192.

Plauger, P.J. *The Draft Standard C++ Library*, Prentice Hall, Englewood Cliffs, NJ 1995.

Martin Carroll, "Invasive Inheritance," *The C++ Report*, October 1992, pp. 34-42.

Cline, M. P. and Lomow, G. A. *C++ FAQs: Frequently Asked Questions*, Addison-Wesley, Reading, MA 1995.