

Object-oriented programming in C

Eric Blair

2 January 2006

Here are notes on object-oriented programming (OOP) in C, aimed at people who are OK with C but are primarily versed in other fancier languages.

It is to some extent a response to those who write C off as obsolete because it lacks the cute syntax of more recently-written languages. Without the cute syntax—with just 21 keywords and a lot of stars—you can still get as much interesting structure as other languages that impose the structure via more keywords or punctuation, and you can often do it in an easier-to-maintain way. So, here's how.

Before I start, I need to give the same caveat I always give when I write about C: I work in what used to be called *computing* and is now called *scientific computing*, mostly in statistical analyses and agent-based modeling [(which the more ignorant insist is impossible in C. Haha, people pay me to do the impossible)]. If you're writing end-user programs with glossy GUIs, you're not gonna write the GUI in C. If you need a shell script, write it in Python. But the computations at the core of it can still be in C, and the methods below may help your C code play nice with your GUI wrappers.

What is this object-orientation? The OO framework is in some ways just a question of philosophy and perspective: you've got these blobs that have characteristics and abilities, and once you've described those blobs in sufficient detail, you can just set them off to go running with a minimum of procedural code. If you want to be strict about it, the objects only communicate by passing messages to each other. All of this is language-independent, unless you have a serious and firm belief in the Sapir-Whorf hypothesis.

Much of object-oriented coding is distinguished via a method of scoping. Scope indicates what, out of the thousands of lines of code and dozens of objects you've written down, is allowed to know about a variable. The rule of thumb for sane code-writing is that you should keep the scope of a variable as small as possible to get the job done. Think of a function as a little black box: you want it to have as few moving parts as possible and to run independently of the outside world.

From the OOP perspective, this translates into dividing variables into private variables that are only internal to the object, such as the internal state of the car's motor, and things that the whole world can use, such as the location of the car. Thus, every OO language I can think of defines `public` and `private` keywords.

But wait, there's more: sometimes, you really have to break the rules, just this once, and check the internal status of the motor. You can make the status variable global, defeating the whole mechanism, or you can define a `friend` function. Below,

we'll have inheritance, and will also need `protected` scope. Sometimes, the `::` operator will get you out of a jam.

That is, we can divide the OOP additions to C's syntax into two parts: syntax to give you stricter, finer control over scope, and syntax to override those stricter controls.

How does C do it? The scoping rules for C are defined by the file. A variable in a function is visible only to the function; a variable outside the functions, at the top of a file, are visible only in that file.

A typical `file.c` will have an accompanying `file.h` that simply declares variables and functions. If another file includes `file.h`, then that file can see those variables and functions.

As for the naming, objects let you name functions things like `move` and `add` and never worry about interfering with fifty other functions with the same names. This is nice, but there is a simple C custom to take care of that: prepend the object name. Instead of the C++ `my_data.move()`, where you just understand that this `move` function refers to an `apop_data` object, you'd have a function called like `apop_data_move(my_data)`. There ya go, crisis averted: no name space clashes. Some readers somewhere may complain that the name-prependng is ugly, or that such naming requires programmer discipline without compiler checks, to which I respond: if you're worried about these things you're even more boring than I am.

But seriously, go have a look at Joel the guru¹ for more on how wonderful naming similar to this can be.

So C already has a scoping system that matches C++ if you use the one file-one object rule and a few customs in naming. Thus, adding a whole new syntax for scoping on top of this is basically extraneous, and could create confusion now that you've got two simultaneous scoping systems in action.

Inheritance and overloading Overloading functions and operators is dumb. Joel's article above has a humorous bit about this, which opens: "When you see the code `i = j * 5;` in C you know, at least, that `j` is being multiplied by five and the results stored in `i`. But if you see that same snippet of code in C++, you don't know anything. Nothing." The problem is that you don't know what `*` means until you look up the type for `j`, look through the inheritance tree for `j`'s type to determine *which version* of `*` you mean, et cetera.

Say you have a `blob` object, and think, faux pas, that it is a `blobito` object. You call the `my_blob.cleave()` function. In C, you would be notified of your error at compile time (you'd be calling `blobito_cleave(my_blob)` when you should be calling `blob_cleave(my_blob)`). In many interpreted languages, you would be notified of your error at run time, or sooner depending on the language. In C++, with appropriately defined methods, you would never, ever be notified of your error. That is, operator overloading allows you to bypass a large number of safety checks. It's probably the case that `blob::cleave` and `blobito::cleave` do somewhat similar things, so the effect on the output may be wrong in subtle ways.

¹<http://www.joelonsoftware.com/articles/Wrong.html>

Option B is inheritance via composition. For example, Apophenia has an `apop_data` type:

```
typedef struct apop_data{
    gsl_matrix *data;
    apop_name *names;
    char ***categories;
    int catsize[2];
} apop_data;
```

In OOP-speak, the `apop_data` structure is a multiple-inheritance child of the `gsl_matrix` and `apop_name` structures (plus an array of strings). All of the functions that operate on these parent objects can act on elements of the child `apop_data` structure, and life is good.

On the one hand, this means that if a function acts on a `gsl_matrix *` you can't transparently call, e.g., `apop_sv_decomposition(apop_data_set)`; you have to know that there's a `gsl_matrix` inside the data set and that's what's being operated on: `apop_sv_decomposition(apop_data_set->data)`. On the other hand, you can not accidentally call the wrong instance of the function and then spend an hour wondering why the function didn't operate the way you'd expected. On the minus side, the internals of the object aren't hidden from you—but on the plus side, things aren't hidden from you.

the void, templates And finally, for when you really don't want to deal with types, there's the void pointer. Here's a snippet from an early draft of Apophenia's `apop_model` type:

```
typedef struct apop_model{
    char name[101];
    apop_model * (*estimate)(apop_data * data, void *parameters);
    ...
} apop_model;
```

Two things to note from this example. First, including a function inside a struct is a-OK. We'll declare something like: `apop_model apop_GLS = {"GLS", apop_estimate_GLS, ...}`; and then we can call `apop_GLS.estimate(data, NULL, sigma)`; just like we would in C++-land.

Second, there's the void pointer at the end of the declaration of the `estimate` method. Here's the declaration for `apop_estimate_GLS`: `static apop_estimate * apop_estimate_GLS(apop_data *set, gsl_matrix *sigma)`. Notice that that third argument is typed as a `gsl_matrix *`, even though we're plugging it in where the template asked for a `void *`. [Non-OOP quiz question for the statisticians: why is this a terrible way to implement GLS?] Other models require different parameters, like the MLE functions take parameters for the search algorithm, but they're also called via the same `model_instance.estimate(data, params)`; form. Notice also how the function pointed to is declared to be `static`, so outside the file it's only accessible as the `object.estimate()` method.

In short, the void pointer is your way of saying "Dear C type-checker: leave me alone." The type-checker will still check that you're sending a pointer and not data, but from there you're free to live it up.

By the way, I can't recall ever using this, but if you wanted to, you could even type-cast inside the function:

```
void move(void *in, char type){
    if (type == 'a'){
        a_type a_in = in; a_move(a_in);
    } if (type == 'b'){
        b_type b_in = in; b_move(b_in);
    }
}
```

The void pointer is how you would implement template-like behavior. For example, here is a linked list library (gzipped source) that I wrote when I was avoiding harder work. It links together void pointers, meaning that your list can be a linked list of integers, strings, or objects of any type. How's that for a nice, concrete example.

This self I've only wanted something like the `this` or `self` keyword maybe twice, but I have no idea how to gracefully implement it in C, if at all. [Maybe with the preprocessor?] So I'm open to suggestions on this one.

Refs More essays along the same lines:

A full book² that goes into great detail about the above simple tricks, and also goes much further in implementing something that looks like C++.

An article that focuses on encapsulation, with some suggestions on hiding data.

Another article that blew way past my attention span, and basically shows you how to write a C++ compiler in C. Given my disdain for overloading and strict inheritance (as opposed to inheritance via composition), I wasn't really into it.

OK, there you have it: most of the basics of object-oriented programming implemented via relatively simple techniques in C. The moral: object-oriented coding is a method and a mindset, not a set of keywords.

²<http://www.planetpdf.com/developer/article.asp?contentid=6635>