Object-oriented programming in C

Ben Klemens

22 April 2010

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.

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 outside-the-blob 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.

Scope 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 just as many exposed parts as are necessary to interoperate with 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 scope, given that it has (depending on how you count) about two keywords for scope control? 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, is 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 well. Thus, the private variables are invisible outside the file, and the public variables declared in the header can be used by the other files where you choose to include `file.h`.

The variables in the header file need to be declared with the `extern` keyword, e.g. `extern float gas_gauge`, which indicates that the variable is actually declared in a single `.c` file, say `dashboard.c`, but another `.c` file that includes this header, maybe `motor.c` is being made aware that there is an floating-point variable named `gas_gauge` declared somewhere external to `motor.c`. As noted, the custom is to put all these `extern`s in a header file and be done with it, but if you want to have especially fine control of scope across files/objects, then you can insert exactly as many `extern` declarations as you need exactly where you need them; similarly for function declarations.

The naming thing 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 with a name like `apop_data_move(my_data)`. There ya go, crisis averted: no name space clashes. Some readers somewhere may complain that the name-prepending is ugly, to which I respond: care less.

But seriously, go have a look at Joel the guru[1] for more on how wonderful naming similar to this can be.

C already has a scoping system comparable to that of C++ if you use the one file-one object rule and a few customs in naming. 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 which includes a `cleave` method that splits the blob in half, and a `blobito` object that includes a `cleave` method that binds together internal elements. You have a `blob` object named `my_b`, and, *faux pas*, think that it is a `blobito` object. You call the `my_blob.cleave()` function, expecting that `my_b.size` will double, but instead it halves.

This may sound like a silly example, but from my experience, the most common use of OO machinery such as inheritance is where two objects are very similar but subtly different. Textbook examples: `accountant` and `programmer` objects that both inherit from the `officedrone` object, or from the U.S., `state` and `district` objects that inherit from the generic `us_division`, and are identical except that the `district` object has no `senators` or `representatives`.

[1] `http://www.joelonsoftware.com/articles/Wrong.html`

Those situations where two objects are similar and therefore easily confused are the ones where we most need a syntax that breaks when we make a mistake in guessing the type. If you were doing this in C, you would be notified of your error at compile time (because 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.

I promised you notes on how C does it, not rants about overloading, so let us move on to Option B: inheritance via composition. For example, Apophenia has an `apop_data` type:

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

[This essay was originally written in January 2006, and Apophenia has evolved and stabilized since then. So the sample code is not apop-accurate, but is still fine for getting across the principles discussed here.]

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. To go further with OOP jargon, C lets you extend a structure via a *has-a* mechanism: we have a `gsl_matrix` and want to give it names, so we create a structure that *has-a* matrix and names. Typical OOP languages allow you to extend via *is-a*, wherein your `named_matrix` *is-a* `gsl_matrix` plus the additional elements to add names. I'm no OO pro, but I think you're supposed to read those sentences like the Italian chef in a Disney movie.

On the one hand, having only *has-a* to work with means that if a function acts on a `gsl_matrix *` you can't transparently call, e.g., `apop_pca(apop_data_set)` [PCA=principal component analysis]; you have to know that there's a `gsl_matrix` inside the data set and that's what's being operated on: `apop_pca(apop_data_set->matrix)`. 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.

So 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 a GLS estimation function as `static apop_estimate * apop_estimate_GLS(apop_data *set, gsl_matrix *sigma)`, declare the model via something like: `apop_model apop_GLS = {"GLS", apop_estimate_GLS, ...};` and then we can call `apop_GLS.estimate(data, sigma);` just like we would in C++-land.

[If you didn't follow the syntax of declaring hooks for functions, see p 190 of *Modeling with Data*.]

Second, there's the `void` pointer at the end of the declaration of the `estimate` method in the structure. Notice that that second argument of `apop_estimate_GLS` 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.

It's up to you, the user, to remember what types make sense for what models, because 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 and/or segfault.

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.

The primary benefit from C++'s template system over using `void` pointers is that the template system will still check types. Personally, I've rarely had problems. If I have a list named `list_of_data`, I know to not add `gsl_matrix`es to it. Not having type-checking means that it's up to me to make sure that the wrong thing is never the intuitive thing to do.

By the way 1: notice how `apop_estimate_GLS` is declared to be `static`, so outside the file it's only accessible as the `apop_GLS.estimate()` method.

By the way 2: 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_move((a_type*) in);
    if (type == 'b')
        b_move((b_type*) in);
}
```

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.

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.

Refs More essays along the same lines:
A full book[2] 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.

[2] `http://www.planetpdf.com/developer/article.asp?contentid=6635`