Using a program as a library

Ben Klemens

13 April 2009

I often have this scenario: I have an analysis to do using some quirky data set. The first step in every case is to write a function to read in and clean the data. Along the way to doing that, I'll write functions producing summary statistics sanity-checking the data and my progress.

At this point I can get to the actual process of producing a descriptive model, and then testing that model's claims. This will all be in `modelone.c`

Next week, I have an idea for a new descriptive model, which will naturally make heavy use of the existing functions to clean data and display basic statistics. So how can I most quickly get to those functions while doing minimal damage to the original program?

In the context of C and many, many other systems, the only difference between a function library and a program is that a program includes a `main` function that indicates where execution should start. So the problem is basically in making sure that at compilation, there is exactly one version of `main` visible to the compiler at a time. Here are a few options, all of which are appropriate in some circumstances, though I'll focus on the last here.

Option one: Simply add more functions embodying the new model to `modelone.c`, and add a command-line switch to select the model.

Pros: immediate (especially if you don't use `getopt` to parse the command line). Cons: gets messy very fast. Something about a single several-page code file discourages reading.

Option two: Move all of the more useful functions in `modelone.c` to a second file, `model_lib.c`, write a header, and then #include the header in both `modelone.c` and the new `modeltwo.c`. Pros: very organized. Cons: can take time to do it right, which may not pay off for an isolated project.

Option three: Conditionally comment out the `main` function. Here is a skeleton for `modelone.c`:

```
void read_data(){
    ...
}

#ifndef MODELONE_LIB
int main(){
    ...
```

```
}

#endif
```

If `MODELONE_LIB` is not defined (note the use of `#ifndef` rather than `#ifdef`), then `main` will appear as normal, so you can compile `modelone.c` as normal.

If that variable is defined, then `main` will be passed over—and suddenly you have a library instead of a program. So `modeltwo.c` will look like this:

```
#define MODELONE_LIB
#include "modelone.c"

void run_second_model(){
    ...
}

int main(){
    read_data();
    run_second_model();
}
```

We've successfully used `modelone.c`, which had been a program file, as a library file.

Pros: you don't have to rewrite `modelone.c`, save for adding the if/endif. Thus, this is fast and doesn't require any re-testing of your original work. Cons: if you left a lot of globals floating around in `modelone.c`, you now have all of those globals floating around in your second model. This will be a good thing for some globals, but side-effects may creep in if you aren't aware of what else you're bringing in.

Adding `main` **to a library**   In the other direction, there's good reason to have a self-executing library: testing. Rather than writing the actual library and then a separate file for testing, just put all the tests at the bottom of the library file, along with a `main` routine to run them all. Here, the default usage is to not run `main`, so surround it with `#ifdef RUN_LIB_TESTS ... #endif`, and then define `RUN_LIB_TESTS` only during testing. You may be able to surround all of the testing functions in the `#ifdef`, so non-testing library users can't see any of the test functions at all.

You can define `RUN_LIB_TESTS` either via a `#define` line at the top of the file that you keep normally commented out, or during compilation, by specifying `-DRUN_LIB_TESTS` among the C flags to GCC (or via comparable means for other compilers).