Interacting with C code

Ben Klemens

17 August 2013

A REPL is the read-evaluate-print loop. This description is from the perspective of the computer, which reads your command, evaluates it, prints the result, then waits for your next command; but it could almost be from your perspective, where you read the output from the last command, evaluate what it's telling you about the analysis, and print/type your next command as a reaction to the results of the last one.

I use a compiled language, C, to do things that are traditionally done in REPL-based scripting languages. The last few episodes showed you that writing in C isn't necessarily more difficult; if you can do an operation like de-crosstabbing a data set in Python in a few lines, then you can probably do it in a few lines in C, provided you have the right library of functions.

But fast writing is only half the fun. The other part is interacting with and reacting to the analysis so far. I'll address my techniques for doing so in this entry.

- I always have a terminal for compilation/running and a text editor side-by-side. I've already blogged about how I use vi, zsh and tmux (entry #151), but you can do this with a friendlier GUI-based text editor and a terminal too. If I'm working in R, I still have R in one pane and a text editor with the script in the other, and my R history is mostly `source("script.R")` over and over.
- The C haters complain about how slow the compile-run loop is, which is completely beyond me. If your program is named `163-de_xtab`, then type `make 163-de_xtab && ./163-de_xtab`, and in the time it takes you to hit <enter>, your program is running (or reporting syntax errors, as you'd get via the REPL). That's the first time; after that, just hit the up arrow and <enter> to repeat that same pair of commands. I really can't count two keystrokes and a tenth-of-a-second wait as serious labor. [Many shells also repeat the previous command line with !!<enter>,which I somehow have grown to prefer. It's fully three keystrokes,but you don't leave the home keys.]
- Apophenia doesn't display anything to screen unless you ask it to.
  - The verbosity setting provides one information source for interaction:
    * setting `apop_opts.verbose=-1` gives total silence,
    * `0` means print only clear errors,
    * `1` means errors+warnings,
    * `2` means errors+warnings+queries+basic debugging info,

    - `* 3` means errors+warnings+queries+voluminous debugging info
  - The otherwise-quietness means that I have to use the `apop_data_print` and `apop_model_print` functions with great frequency, but see below.

So for the easy stuff, I am incrementally building the program in the text editor and rerunning it after every new thought gets added. The tail of the script at the moment has a lot of little print statements and a raised verbosity level, which will eventually get dropped when I'm satisfied with the results there. This is basically the REPL form with a solid, executable transcript of everything that I need to get to this point.

The setup is not C-specific, and is a good reminder that there are two types of command that we give our computers: worth recording, and completely disposable. Just the other day, a coworker was complaining about how she spent a lot of time rederiving the commands she'd used for important results, because the REPL encouraged her to mix the important stuff with the trivial little checks and digressions; next thing you know important steps of her analysis are living only in her command history. If an analysis isn't replicable, did it really happen?

This is subjective, but I think that REPLs also encourage thinking of a single line as a unit of thought. Users are focused on composing the next statement in the conversation, or modifying the previous single statement to improve upon it. It's difficult to write a several-line function via REPL. Writing on the full page of a script in a text editor makes us a little more likely to think in terms of paragraphs and functions.

The debugger But that's just for the quick-and-easy scripts. Most of the time, I'm working from the debugger, gdb.

You can write macros for gdb. Here's the one I depend on more than any other (where `p` means evaluate and print the expression):

```
define pd
    p apop_data_show($arg0)
end
document pd
Call apop_data_show to display an apop_data set.
end
```

Put this macro—and more macros to display any other nontrivial data structures you work with—in `~/.gdbinit`. Now I don't care that data sets don't print to screen by default, because I can use `pd` to print them at will. Using the example from last time, I might

- start gdb with `gdb ./163-de_xtab`, then,
- now that I'm at the gdb prompt, set a breakpoint at line 5 with `break 5` (or `b 5` for short),
- run the program with `r`,
- and, once it stops just before executing line 5, display the data set `d` with `pd d`.
- Maybe I'll go further, and call another function, like `pd apop_data_summarize(d)` to get means, medians, standard deviations, and so on (though you'll really want `pd apop_data_summarize(apop_data_transpose_base(d,0))`).

Which is to say, gdb is my REPL. Except REPLs are always at the tail-end of the script, while the debugger encourages users to stop at arbitrary points along the way, ask questions, then keep going. [R users, you can use the `browser()` command to do something similar, though it's no surprise that it's a little limited compared to using a full, dedicated debugger.]

My screen looks similar to before: gdb in one terminal, and the text editor in another. I've tried several IDEs (integrated development environments) that merge the debugger and the text editor, and always find them to be a little more awkward than the simple two-terminal solution.

This is not without flaws. Notably, gdb doesn't know about macros. The `apop_data_transpose` function is actually a macro, which puts named arguments in their correct slots and inserts default values in those arguments to the base function that you don't specify. gdb doesn't know the macro, so you're stuck with using the base function and all its arguments. `apop_data_show` is not a macro, so you can use it in the debugger.

A good compiler is aggressive about optimizing out variables and even commands, so I am sometimes frustrated that I can't get the value of a temp variable because it doesn't exist anymore; then I have to go to the makefile and turn off optimizations (remove `-O3` or change it to `-O0`). So it goes.

You don't have to leave gdb to compile, by the way: you can run `make` from the gdb command prompt, and I've even aliased

```
define mr
    make
    run
end
```

to keep that sequence down to two keystrokes plus the <enter> key.

And one last macro:

```
define query
    p apop_data_show(apop_query_to_text($arg0))
end
document query
Query the database Apophenia currently has open. E.g., query "select * from sqlite_master" to
     view the database schema.
end
```

By the way, R users, you can debug the C side of your packages via `R -d gdb`, which puts you at the gdb prompt, where you can run R using the same `r` command that starts any other program. Once things get running, you can set breakpoints in your C code as usual, and can use the `backtrace` command (`bt` for short) to see your C code at the top of a stack of thirty or forty frames built by R.