

Variable Declarations

Eric Blair

11 November 2003

I acknowledge that I'm going to alienate many of you here, for which I apologize. Except for a distasteful meat-eating metaphor and a claymation cartoon, this one won't be much fun for non-academics. I promise next time will be better.

That said, I've decided on a programming language for me. Let me explain the problem for those of you whom I've just alienated but are still reading anyway: there are several hundred different programming languages out there, and the first, angst-inducing step in beginning any project is committing to one. The genesis of the problem is that if you're a computer science student, and you need to do some dumb little project, then of course you're gonna purpose-build a programming language in which to do the project. And so, Google has a list of about 170 languages, and much electronic ink has been electronically spilled on why any is better than any other.

I'm mostly thinking about modeling and statistical applications. Statistics packages are basically scripting languages: excuse my abruptness, but if you don't have a script which lists exactly what you did to go from raw data to the output you put in your paper, then nobody has any reason to trust your results. The interactive interface is nought but a convenience to help you write that script.

So, without further ado, let me tell you the modelling/stats language for me: C. It took me years to work this one out, but at this point, I've forgotten why we've been avoiding it. I mean, it's a big winner on almost every count that we use to score scripting languages: the code base is gigantic, it's eminently portable, and it's the fastest of any of the languages—because almost all of the others are written in C, so the computer often has to translate from the language of the month to C before compiling the program.

[Human interest part: C was written in 1972, as a successor to the B programming language (which was successor to the boringly-named BCPL). It was partly written to be used for writing the UNIX operating system. At this point, I think there's approximately zero benefit to the scripting languages. Literally anything you can do in these languages, you can do in C, both in the sense that many of them are just shells to C itself, and that they're all Turing complete anyway. [Translation: Alan Turing described a very simple theoretical universal computer, which does 100% of what a modern computer can do. All of these languages are equivalent to that theoretical machine.]]

I think this is me maturing here. I've begun approaching every project as one which is not a quick answer to a quick question, but one which will haunt me for a year or more. As such, the post-C scripting languages are sort of a burden.

The first reason why there are so many post-C languages in use is the pointer issue. See, on top of the usual variables that you care about (integers, characters, real

numbers), C has variables to represent the location of a given integer, character, or real number in the computer's memory. But when writing a program, people don't want to have to think about the location of their numbers in computer memory, just as when carnivores eat a sandwich, they don't want to think about the location in the cow where their piece of cow came from. So the first nice thing scripting languages do is hide the pointers.

Beyond that, they all have certain cute features that make it easy to write ten lines of code that do something—but then you add a few lines a day until the program is huge and you would've been better off if you'd done it all in C, but now it's too late and the cute initial features have just become an albatross around your neck, causing you to slouch still more than you normally do when you're programming. The most obvious example is the lure of not having to declare whether your variables are integers or reals. In your first half-hour of programming, it's sort of convenient, and one less thing to think about while you're throwing together your little procedure. But then the program is going to guess at whether x is an integer or a real every time it deals with it for years to come. For some situations, this can be hundreds of thousands of times, and it'll guess wrong a few times, and this will all translate into real human frustration.

E.g., I was in love with R for its attribute structures, which let you name the columns of your arrays of mixed data types, and for saving me the trouble of having to declare all these complex types. But then it kept coercing my variables into the wrong types all the frigging time, and I kept having to work around that. I've translated my whole migration simulation to C, and it runs literally a hundred times faster, and is only about 10% more lines of code. I can tell you similar stories (some hearsay) about Octave, Matlab, and Java.

Also, when something goes wrong, there's no doubt that it's my fault.

Now, there are situations where other programs are way better than anything you could do yourself in C, but there are a lot less than 170 of these situations.

- *R* has a neat-o structure which embodies a statistical model. In some cases, it does more than just solve $(X'X)^{-1}(X'Y)$.
- *Lisp* lets you compute on the language (i.e., write lisp code that is then evaluated as part of your program). I don't know the best language in the lisp class, but acknowledge that some folks need this stuff.
- *Mathematica*, Matlab, and Mupad have built-in lists of a few hundred transformation rules for symbolic algebra. No way you're writing that in C. But—and I can't stress this enough—if you're not doing symbolic algebra, Mathematica has no real advantages over other programming languages.
- *Perl* is good for those dumb little shell scripts, and it has regular expression handling which is above and beyond the standard C library, and we all know how important good regular expression parsing is (but that's another blog. Anybody out there ever use Monarch?). On the other hand, it's no surprise that Larry Wall, two-time winner of the International Obfuscated C Code Contest (1986 & 87) came up with Perl, which is often described as executable line noise.

In terms of the ‘does it have something exceptional that you can’t do in C’ test, Matlab, Octave, Gams, Gauss, Sas, Spss, Stata, all have nothing (see below about the pretty pictures).

Biting the bullet So here’s some advice in the imperative tense, based on what I’ve learned about C programming in the last few weeks, for the reader with lots of scripting experience but not much C experience.

First, take some time to get familiar with pointers. Here’s a claymation cartoon to help start you off. Google will give you a few dozen more thorough tutorials.

As for choice of compiler, use GCC (which stands for either the GNU C Compiler or the GNU Compiler Collection). It’s free, portable to *everything*, and has been debugged and optimized by hundreds upon hundreds of people. If you pay a few hundred dollars for SunOS, this is the compiler they’ll give you. Oh, and the nice people at GNU have also written a C tutorial to get you started. [html¹ or PDF²]

Get to know the libraries. The standard library has 90% of the wheels you’re likely to reinvent, and then there are heaps of other libraries³ for the less standard stuff. [The GNU scientific library is especially notable here. You can think of Matlab as a shell for the GSL]. The documentation for the standard library is very readable, and should just be left open while you’re writing any program.

A common complaint about C is that since it’s compiled, you can’t have interpreted-language fun, typing in variables willy-nilly and seeing what happens. But you can—just use the debugger, GDB. Getting to know GDB is really the one thing that’s most made programming in C as fun and easy for me as using a scripting language.

I’ve been using kdbg as a graphical interface to gdb, but have found it to be extraneous much of the time. Here⁴’s a list of others. You creative types can probably configure one of these things to feel like the Matlab GUI.

Not as essential, but nice, is revision control. This one is in no way C-specific: you could use it for any projet, including your dissertation and/or Great American Novel.

Graphics: This is where a high-level language is actually useful. Do the work in C, then output the results to a file and use one of the cute scripting languages I’ve spent the whole column ranting about to produce the final pictures. Same with the other packages with exceptional features above: get in, use the feature and get out. Just because you can write the whole program in the package doesn’t mean that it’s a good idea to do so.

¹<http://www.crasseux.com/books/ctutorial/index.html>

²http://www.eng.usm.my/~badli/eeu101/gnu_c_tutorial.pdf

³<http://www.gnu.org/directory/libs/>

⁴<http://www.gnu.org/software/gdb/links/>