

How to free yourself of patriarchal fetters

Eric Blair

24 July 2005

[A few grad students have asked me to send them some notes on making the transition from a higher-level language like Matlab to C. So, here it is, for the public record.

There's some overlap between this and my in entry #011, but Ms EB of Washington, Columbia, an Ashtanga Yoga teacher, suggests that I provide more repetition, so here goes. I'm often surprised by how little time I spend talking about C, given that it's what I spend most of my working hours with.

I'm not gonna talk about whether C is right for you, blah blah blah. If you're doing a large-scale simulation, do it in C. For organizing your porn collection, there are other languages.]

The best way to explain how one would transition from a high-level language like Matlab to C is to explain how C is different. The key differences are in the management of memory and in the tools which support C; beyond that there are loads of boring little details. Let me begin with the memory management issue, because it motivates why you might want to learn C to begin with.

The pointers The thing that *really* distinguishes C from virtually every high-level language is that C gives you two means of dealing with memory, which you can think of as automatic mode and manual mode.

In manual mode, you declare that some piece of memory somewhere will hold your variable, and then you get a pointer to that piece of memory. Think of a little garden plot in the middle of nowhere. You don't really care to visit it yourself, but you have a robot assistant you can send out to tend the plot for you. When you send it out to add fertilizer, you'll need to make absolutely certain that you're telling it to dump it in the right plot of land (because outside of the garden, we don't call it fertilizer); if you lose the slip of paper where you wrote down directions to the garden, you can kiss your tomatoes goodbye; if you accidentally tell your robot assistant to plant cucumbers in exactly the same spot, then it'll trample the tomatoes and instead of a tomato plot plus a cucumber plot, you'll just have a cucumber plot and salsa residue.

There are a lot of things that could go wrong, so the stats packages help you deal with manual memory allocation by not letting you do it. 100% of memory handling is automatic, and now you can't hurt yourself with those sharp pointers. They're probably right: if I'm a student who just wants to do his matrix algebra homework, I could care less about memory management. However, if you're doing a large-scale simulation, manual memory management becomes essential.

Let's say you write a function to shuck corn. With the automatic memory allocation paradigm, you have a big pile of corn at your feet. You know exactly where it is, so it's easy to call the `corn_shuck` function, by hiring a truck to send your pile of corn to the function. Conversely, under the manual memory management paradigm, all you

have is an address, so you send the function a note saying ‘please go to my garden at 0xbb88d345 and shuck whatever corn you find there.’ You could get the address wrong or forget to plant the corn, but if you do it right, sending a note with an address is a much more efficient process than sending a truck.

OK, enough with the metaphor: when you call a function with automatically allocated memory, you send in a *copy* of the data, which may entail some significant labor in making that copy; when you call a function with manually allocated memory, you just send in a copy of the address, which just requires copying a single number. I have some functions which are called literally billions of times over the course of the simulation over on the other screen; in a system which insists on copying in a structure every time, my two-hour run would take weeks to finish.

Pointers are cognitively difficult, meaning that unlike the semi-memorization involved in learning syntax details, you will have to learn. You will confuse yourself repeatedly about whether you are referring to an array element or the array itself or a pointer to the array. The syntax won’t help, because the declaration of a variable and its use both involve stars, but in different, incompatible ways. The compiler will help, because a `char*` and a `char` are different types, so the compiler will yell at you when you screw up.

Manual memory errors are confusing because when the robot servant brings back cucumbers instead of tomatoes on line 3000, it could be because you had sent it out to plant cucumbers on line 2000; stare at line 3000 all you want, but you won’t find your bug. Because tending manual memory is known to be difficult, there are loads of utilities to help you along. The debugger will be invaluable. If you’re lucky, Valgrind will work for you. There’s electric fence, by a certain open source talking head.

The workflow Matlab is an integrated development environment (an IDE), meaning that you never leave the Matlab window to do anything—and golly, you can’t, since nothing but Matlab (and Octave) can understand Matlab code. On the other hand, C has been a standard for a few decades now, and that means that there are an absolutely overwhelming number of programs that will help you write C code.

You’ll need a good text editor in which to write the code. There are many with useful features for writing code like color syntax highlighting; at the least, you need one that will let you jump to line 267 quickly.

Then there’s the compiler, which translates your text into something the computer can run. Next—and this is not optional—is the debugger, which you’ll be spending a lot of time in.

There are IDEs for C which will gather all this together for you. I’ve never dealt with them, so I can’t advise you. Ask Google. Oh, and see also the ddd front-end for gdb.

The typical routine goes like this: you have the text of your program in one window, the compiler in another, and the debugger in a third (and documentation in your browser). You write a few lines of text, then compile. The compiler yells at you that you’ve gotten some details wrong on line 267; you go there and end your paren or add your semicolon. When the compiler is happy, you run it under the debugger, at which point your program will run briefly and then crash. You get a trace at the point where

it halted, and then go back to your code and fix your error. Repeat until it runs. If the answer looks completely wrong, add breakpoints in the debugger so you can observe your intermediate results. Keep running and modifying until it looks right.

This probably sounds a lot like your current workflow, except it's much more decentralized. You'll also be spending more time in the debugger. [Life is just too short to not be using a debugger all the time. If something breaks, eyeball your code for a few seconds, and if it ain't obvious, go straight to the debugger. If your current favorite language doesn't have one (you may need to search the 'nets for it), ditch the language.]

The libraries Matlab has a whole lot of functions built in, either to the syntax itself (like matrix multiplication) or as functions they wrote for you (the .m files). C has next-to-nothing built in to the language, and therefore depends heavily on external functions.

You're given the standard library by default; here's the standard library documentation. Beyond that, there's the additional stuff for the work specific to what you're doing. The range available is stunning. You will almost certainly want to pick up the GNU Scientific Library; I've written a library on top of it to facilitate some things that I found in need of facilitation, in which you may or may not be interested.

This is where `make` also comes in handy, because you now need to link together your program with these external libraries, and the linking commands get long; `make` will help you organize them. This will involve some frustration as you try to get the link flags right in your Makefile, but once you're set up, you'll never think about it again, and this host of useful functions will work seamlessly into the rest of your work.

Syntactic sugar Matlab and Co. do a lot behind your back, which is why they are easy to start with and which is why they are slower in the long run. So you will now need to do a lot of boring coding yourself. Notably, you need to declare your variables beforehand. People whine about this all the time, but it's just a piece of tedium that you'll get used to. The compiler will warn you when you haven't declared something.

C just has no glamour to it. It requires that you cross all your own *ts* and has no specialized *i*-dotting syntax to make the code look more like an equation. There's no cute syntax to operate on every element of a vector at once—to operate on a matrix, you'll need to step through every element yourself. This is boring and inelegant. Deal with it. [Some people go further and imply that languages which deal with vectors directly are somehow technologically superior, but this is just dumb. These languages may have a few tricks at hand, but in the end they have to iterate item by item for you. Some processors like the PowerPC have a vector type, but unless you're using a vector-based processor, you're stepping through item by item no matter what the syntax looks like.]

Learning C OK, so this should give you a good idea of what you need to focus on to transition from more patriarchal languages to C. You'll first need to get the assorted parts: `gcc` (the compiler), `gdb` (the debugger), `make`, the GSL, maybe `valgrind`. Lucky I gave a summary in entry #137 about how to use package managers to get heaps of software at once.

Once your toolchain is in place, you'll need to get to know the tedious syntax, about declaring things and writing for loops correctly. This is easy; just a question of developing some habits. Type 'C tutorial' into your favorite search engine and start reading. Modeling with Data also has a tutorial which is more conceptual and less nuts-and-bolts.

Part of the process is getting to know the linker and the debugger. This means reading more manuals, but nothing really difficult. Most high-level languages include a debugger, and all debuggers have the same set of concepts, so you can futz around with one in your familiar setting before dealing with gdb. The linker thing is totally absent from most packaged languages, and will annoy you until you learn the difference between `-l` and `-L`.

Finally, when everything compiles and the environment feels reasonable, you should spend a few hours making a concerted effort to get down the pointer thing. Again, the search engines are there for you, and there are more links from my first post on C, linked at the top of this column.

You can see that there's a lot of front-work before you can get stuff done—especially now that the typical computer lab computer doesn't have a C toolchain installed by default. Leaving behind the fetters of the packaged language requires taking on responsibilities you'd rather not care about, but for computationally intensive work, the freedom is worth it.