acmqueue

Porting with Autotools

Using tools such as Automake and Autoconf with preexisting code bases can be a major hassle.

Dear KV,

A piece of C code I've been working on recently needs to be ported to another platform, and at work we're looking at Autotools, including Automake and Autoconf, to achieve this. The problem is that every time I attempt to get the code building with these tools I feel like a rat in a maze. I can almost get things to build but not quite.

There must be some rational way to deal with this.

Auto'd Out

Dear Auto,

I am deeply confused why, if you wanted a rational way to deal with something, you have come to me for advice. GNU Autotools are OK for small projects or those that adhere to the tools' way of thinking about code, but they are often quite difficult to work with if you have a preexisting code base. When I say difficult, I really mean maddening. And when I say maddening, I'm really saying something far stronger, but which I'm sure ACM would not want in print. I find the Autotools suite to be one of those things that might have been a good idea at the time but that has grown to be so unwieldy that no programmer I know wants to use them, let alone rework one of their own projects with them.

The most common way to make code portable is to litter the source with all kinds of macros that abstract away the differences between different platforms. The most naive way to do this is the classic

```
#if defined(platform)
#endif /* platform */
```

meme, whereby the programmer brackets the nonportable code with these macros. If you're going to go down this path, then at least do the next programmer the courtesy of properly labeling your `#endif` statement with a comment that says which `#if` the `#endif` matches. If I'm the next programmer to look at your code, that comment just might save your life.

If your code has to support only two platforms, then the bracketing macros are not the worst possible choice to solve the problem; but once the code is made somewhat portable, it would be foolhardy to bet against someone asking for a third platform. I've seen code bases that supported more than a dozen different hardware architectures using macros in this way, and I can tell you that I hope never to see that again. The only way to understand code that has been bracketed in this way is to run it through the preprocessor so that it spits out only the bits of code that it's really going to compile. Attempting to debug source code with dozens of different bits of bracketed code is maddening. For those who are reading this and thinking that this is a problem only in C, I've seen similar types of portability shims in C++ as well.

Of course the correct way to make a piece of code portable is to add an abstraction layer that hides the nasty nonportable bits underneath shiny, clean functions that most of the code uses to get its work done. In fact, this is how the code should have been written in the first place, but of course this rarely happens in practice. There is a reason that programmers created libraries to abstract away the nastier bits of low-level code, and it's unfortunate that people who use libraries never consider writing some themselves.

An easy rule of thumb is that if you find yourself writing the same piece of nasty code twice—and by nasty I mean a piece of code that when you're finished with it you wish never to have to look at again—then it should be placed into a function with some reasonable arguments so that if you wrote that nasty bit correctly, you really will never have to reimplement it, or even revisit it. It's said that Newton discovered calculus twice, but I bet he wished he hadn't had to do that more than once.

As for Autotools, you can spend weeks trying to make your code work with them. I don't know that they will make your code more portable to other platforms, but if they help to remove the tedium of building your code on multiple systems, then you should use them. Me, I'm sticking to my makefiles and abstractions.

KV

Dear KV,

I am working on an app for the iPhone and am blown away by Apple's concept of model, view, and controller, which is how all iOS applications are meant to be written. Why isn't more code written in this simple and easy-to-understand way?

Viewed, Modeled, and Controlled

Dear Viewed,

While I'm sure that the be-turtlenecked one in Cupertino is thrilled that you have drunk his Kool-Aid, Apple did not invent the model-view-controller concept. The idea that software—and in particular software with a complex user interface—should be broken down such that the user interface is divorced from the underlying data has been common in good software design for quite a while.

The reason it may not be familiar to more programmers is that code is often designed from the user-interface point of view, which is exactly the wrong way to go about implementing a piece of software.

How the software looks to the user is an important concern, but it should never be the first concern. The first concern of a programmer really ought to be data. Programs exist to manipulate data: whether that's money in your bank account, comments on a Web site, or characters in a game, the data is the most important thing—in fact, it's really the only thing that the program is ever going to deal with.

The second concern a programmer ought to have is how the data is manipulated, which is where the controller part comes in. How do I translate or transform my data in order to carry out the computation that is the reason for this program's existence? If you get the model and the controller correct, then you can implement just about any user interface you like on top of the first two pieces. In point of fact, hiding the data and having a good set of APIs to access and transform the data is how the best software is written. Having a good set of APIs to manipulate your data makes it far

easier to write command-line interpreters, scripting languages, and graphical user interfaces to manipulate the same data. If the user interface changes, there should never be any need to change the underlying data model.

You might have to add some parts to the controlling code: perhaps a new API that retrieves the data in some different way for a new user-interface widget. Only when a new piece of data needs to be manipulated would the data model have to change.

Only after the first two concerns are dealt with should you be worrying about which bits of bling you will be gluing onto your application. At this point you can get into my favorite discussion with marketing, the one that I remember from long ago when I still thought that working on UIs would be fun: Should the button be red or blue?

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2011 ACM 1542-7730/11/0300 \$10.00