Kode Vicious Battles On

Kode Vicious is at it again, dragging you out of your coding quagmires and kombating the enemies of kommon sense. It sometimes gets ugly down there in the trenches, spelunking the dark caverns of unreadable code and spurious logic, but, hey, somebody's gotta do it.

Dear KV,

I'm maintaining some C code at work that is driving me right out of my mind. It seems I cannot go more than three lines in any file without coming across a chunk of code that is conditionally compiled (i.e., bracketed by `#ifdef/#endif` macros). All of these code blocks make reading the code difficult and debugging almost impossible, and the code has a lot of bugs. Have you come across this before? My temptation is just to rewrite the whole lot of it, but I don't have the time for that—I'm under a deadline, of course.

Sincerely,
At the end of my ifs

Dear Iffy,

KV can, alas, sympathize with your position. While a judicious `#ifdef/#endif` block can be helpful in controlling what pieces of code wind up in a finished binary, overuse of those macros makes code nearly unreadable, and I hate unreadable code.

In most cases the real source of your problem is a failure of abstraction. In C, as in many languages, we have these things called functions, and the reason to have these things is so that at any level of a program the necessary details, and no more, are evident. That balance is not simple to achieve but it is definitely not helped by spreading a bunch of `#ifdef/#endif` clauses over the code.

There are three main reasons given by those who prefer conditional compilation over the use of a function call or a proper abstraction.

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

1. Code with conditionally compiled blocks is faster because it doesn't incur the overhead of a

function call. Once upon a time this might have been true, but that time is long past. The compiler is a lot smarter than you are and knows a lot more about optimizing the code than any engineer futzing with conditional compilation. Instead of futzing with conditional compilation, you should be concentrating on making the code readable.

2. Conditionally compiled code is smaller because it allows you to pick and choose which bits of code are in the final executable. This is another completely unnecessary optimization, and not because of Moore's law or the cost of RAM. There is enough bloatware in the world, and KV does not want to encourage any more, thank you. You see, there isn't just a compiler involved in building your program; there are these other components, called the linker and the loader. The linker's job, as its incredibly transparent name suggests, is to link together your program from all the necessary bits into the final executable. Ever since libraries were invented, perhaps 50 years ago, the linker has been necessary to make a program written with more than one file actually run.

The linker has a companion, which is the loader. While the linker does the work of finding all the necessary function calls and library routines for your program, it is the loader's job to actually get all of this into memory. So, great, you say, now I can abstract all my software and get rid of those `ifdefs`, but my code is now completely bloated. Not quite. Through the miracles of modern computer science research, we have recently been given the great gift of dynamically linked and loaded libraries.

Another transparently named concept, the dynamic linker/loader does just what its name implies. It links and loads code dynamically, at runtime, right before your eyes. With a set of dynamically linked libraries your executable will be only a small component of the system, and if properly abstracted—there's that nasty word again—then only the absolutely necessary set of code will be in memory at any time. Sure, you have to ship the same amount of code, but you can't complain that you

have to load it all into memory at startup.

3. We use conditional compilation to manage our build system to include and exclude functionality. KV has worked on a system like this. The features in the final executable were controlled by a header file with tons of `#define/#undef` clauses that turned features on and off. At first it looked clever, until you realized that the amount of knowledge it took to use the system was too large for most people to handle.

One problem with this system was that using `#ifdef/#endif` hid the interdependencies between modules. Sure you could remove `FOO_FEATURE` by undefining it, but you then might not be able to build the system because of `BAR_FEATURE`, and since the whole system was written by hand it was necessary to remember this yourself. As the system grew to encompass more features, the combinations grew out of control to the point where a tool had to be written just to manage the `#ifdefs`. The problem was that the tool could never be smart enough to do the work correctly, and there wound up always being bits that were missing or wrong, which led to hard-to-diagnose bugs.

Refactoring the code, which is just a nice way to say “rewriting” on a smaller scale, will probably make your life easier.

In the end it was necessary to write a new system that used the linker to find out about interdependencies and express them to the person configuring the system.

Another fun problem was that, of course, people added hidden assumptions when they added code because they had never tested the system with a minimal configuration. Many features were developed while assuming that whole removable subsystems just happened to be in the build. Then someone in the field would try to configure a more minimal system only to find that two large subsystems—for example, the networking code and the Java Virtual Machine—were glued at the hip. This generated meetings that were not fun to attend and required point releases and wasted engineering effort.

Now that I’ve gotten some of that out of my system, I can still feel some bile trying to break free, but let me come back to a more specific answer for you. Look at the code, see which conditionally compiled clauses can actually become function calls, and then make the

proper abstractions. I know a full rewrite would take too long—they always do—but refactoring the code, which is just a nice, politically correct way to say “rewriting” on a smaller scale, will probably make your life easier.

KV

Dear KV,

We’re building out a new Web service where our users will be able to store and retrieve music in their Web accounts so that they can listen to it anywhere they like, without having to buy a portable music player. They can listen to the music at home with a computer hooked to the Internet or on the road on their laptop. They can also download music, and if they lose it through a problem with their computer they can always get it back. Pretty neat huh?

Now to my question. In the design meeting about this I suggested we just encrypt all the connections from the users to the Web service because that would provide the most protection for them and for us. One of the more senior folks just gave me this disgusted look and I thought she was really going to lay into me. She said I should look up the difference between authentication and encryption. Then a couple of other folks in the meeting laughed and we moved on to other parts of the system. I’m not building the security framework for the system, but I still want to know why she said this? All the security protocols I’ve looked at have authentication and encryption, so what’s the big deal?

Sincere and Authentic

Dear Authentic,

Well, I’m glad they laughed; screaming hurts my ears when it’s not me doing the screaming. I’m not sure what you’ve been reading about cryptography, but I bet it’s some complex math book used in graduate classes on analysis of algorithms. Fascinating as NP completeness is, and it is fascinating, these sort of books often spend too much time on the abstract math and not on the concrete realities of applying the theories in creating a secure service.

In short, authentication is the ability to verify that an entity, such as a person, a computer, or a program, is who or what they claim to be. When you write a check, the bank cashes it because you’ve signed the check. The signature is the mark of authenticity on that piece of paper. If there is a question later as to whether you actually wrote me a check for \$1 million—let’s say if I decide to deposit it in my bank account—then the bank will check the signature.

Encryption is the use of algorithms, whether they’re

implemented in a computer program or not, to take a message and scramble it so that only someone with the correct key to unlock the message can retrieve the original.

It's pretty clear from your description that authentication is more important to your Web service than encryption at the moment. Why is this? Well, what you care most about in your situation is that users can listen to the music they've purchase or stored on the server. The music does not need to be kept secret because it is unlikely that someone is going to steal it by sniffing it from the network. What is more likely is that someone will try to log into your users' accounts to listen to their music. Users will prove who they are by authenticating themselves to your service, most likely via a username and password pair. When users want to listen to their latest purchases, they present their username and password to the system in order to get access. There are many different ways to implement this, but the basic idea—that users have to present some piece of information that identifies them to the system to get service—is what makes this authentication and not encryption.

The password need not be encrypted, only hashed, before being sent to the server. A hash is a one-way func-

tion that takes a set of data and transforms it uniquely into another piece of data from which the original cannot be retrieved by anyone, including the author of the hash function. It is important that the hash function produce unique data for each input, as collisions make it possible for two different passwords to be the same hashed data, and that would make it harder to differentiate users.

There are plenty of books and papers on this stuff, but try to avoid the pie-in-the-sky stuff unless you're researching new algorithms, because you really don't need it, and it'll just make your head hurt.

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 has made San Francisco his home since 1990.

© 2005 ACM 1542-7730/05/0400 \$5.00