Gettin' Your Kode On

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

Another year is upon us and we are happy to have Kode Vicious still ranting against the ills of insecure programming, insufficient commenting, and numerous other forms of koding malpractice. Yet despite his best efforts, the bittersweet truth is that these problems are not going away anytime soon, and therefore should continue to provide ample fodder for future KV columns. Oh, to live in a world that doesn't need KV's advice—or doctors, for that matter.

Dear KV,

Simple question: When is the right time to call the `c_str()` method on a string to get the actual pointer?

Hanging by a String

Dear Hanging,

Sometimes I wonder if people are sending me questions just to test me—and I don't mean to test my knowledge, but my patience. In my opinion (note that I left out the *humble* part), `c_str()` must be called only as a last resort and only to call a function that absolutely requires a pointer to memory—for example, sending the string over a network socket with the `send()` system call.

Your letter brings up a broader point, though, about people who seem to work very hard to defeat the APIs they have been handed. I was not present when the C++ string class was written, but I would like to think that one of the motivations for creating the class was, in some small way, to atone for all the problems caused by the unsafe way in which strings were implemented in C. (See KV on `strcpy()`, *ACM Queue*, November 2005, for my comments on the problems inherent in C strings.)

Now I'm sure you, Hanging, would never do anything as completely heinous as using `c_str()` to get a pointer

to the string and then go copying, modifying, or searching the string in the old C-style way, because all of those functions are

provided for you by the class itself. Many people have gone to a lot of trouble to design, implement, test, and optimize those methods; you wouldn't just ignore all their hard work now, would you?

The problem is that many people do just that, and not just with the string class but with all kinds of classes. Instead of extending either a class or library to include their own special cases, they just go around the API, making their code harder to maintain and giving me the high blood pressure for which I'm now in trouble with my doctor. Luckily, they now have chewable Valium; unluckily, my doctor says I should meditate more instead of treating these problems with drugs. Clearly, I need a new doctor.

To sum up, if you absolutely need `c_str()` to call some low-level function that you cannot pass your class to, then that is probably acceptable. Use it for anything else and if I have to fix your code, you'll be hanging by more than a string.

KV

Dear KV,

I've seen you write a lot about C and C++ and the problems those languages have, and you've once referred to PHP, if I remember correctly, but I'm wondering why you don't just advise people to switch to a language like Java, which does not have the pointer safety issues. It's the 21st century, after all; surely we can do better than C and C++.

New World Man

Dear NWM,

I am often amused how each new language generates its share of devotees—those who believe that the new thing is *the* thing. The reason I don't write much about Java is that I haven't gotten many questions about it, which perhaps means it is, as its devotees claim, the be-all and end-all. When interacting with such people, I am often

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!

reminded of Lisp hackers—not all Lisp hackers, of course, but only that small group who believes the whole world should be built out of Lisp code, ignoring all the other things that are around to be used and reused. I recently had a discussion with a Java programmer who, when asked why he didn't just link his code with a C library,

What language you use has very little to do with the quality of the code.

said, “No, no, we'll have to reimplement the library in Java.” At that very moment I wanted to reply with gunfire, but, again, those pesky company rules prevent me from keeping a gun and ammo in my desk, and by the time I could have procured a gun and ammo I probably would have been a bit calmer. Clearly, those HR folks know what they're doing when they write those rules!

All of which brings me to the fact that what language you use has very little to do with the quality of your code. For example, the pointer problem is just one issue in safe programming—and one that gets a deserved amount of attention. But even if this problem is completely solved, there are plenty of things that can trip your program up and make it insecure—for example, the dangerous if clause, which is something I see, unfortunately, every day.

A dangerous if clause is one in which the code you want to protect with the if isn't really protected. Consider the following pseudocode:

```
0: if (out < 0)
1:   return (fileError)
2:
3: if (permission < operator)
4:   return (permissionError)
5:
6: if (data.len() <= 0)
7:   return (dataError)
8:
9: write(out, data, data.len)
```

At this moment you might shake your head, or violently bang it against your desk as I often do, and wonder how such travesties come to exist. Well, this is a case of creeping software crud. Very likely the code was originally

written with only line 9. Later, a bug was found because it was possible for this function to be called with a data structure that didn't have any data in it, a length less than or equal to zero, and lines 6 and 7 were added to address that bug. In some new release it was decided that only a user with a permission of operator or higher could actually use this function, and lines 3 and 4 were added as well. Finally, a bug was found that made it possible for a bad file descriptor to get into the function, at which point lines 0 and 1 were added.

There are several problems with this piece of code. The first is that you can accidentally perform the dangerous function of calling write() if any of the preceding conditions are in error. For example, if a bad descriptor can also be 0 as a result of an update to another library, or if the permission system is changed in some way, it might be possible to call write() when you wouldn't want to. The reason that all the if statements were added was to protect the program from calling the write() function when there was a problem, so the code should be structured in just that way:

```
0: if ((out >= 0) && (permission >= operator) && (data.len() > 0))
1:   write(out, data, data.len)
2: // Put all the error condition returns here.
```

In this version the only way in which the dangerous call will be executed is if, and only if, all of the preconditions are met simultaneously. The write() call is no longer left hanging in the wind. If you still want to differentiate the error conditions, that can be done either in an else clause or as the ending of the function.

The number of times I have come across code similar to the former example and had to make it look like that in the latter is unfortunately very large—not as large as my usual bar tab, but significant, nonetheless.

So you see, NWM, there is still plenty to be done, even when you eliminate the dangers of pointers.

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.

© 2006 ACM 1542-7730/06/0200 \$5.00