Really, Really Going Native

We've been hearing how small ubiquitous devices and vast web sites have reminded everyone that efficiency and performance really are important in code. All I can say is, yes. And we've also have been hearing that as a result, C++ is coming back! Um, why?

The conventional wisdom ten years ago was that size and speed didn't matter much, since desktop hardware was so big, fast and underutilized. Instead, time to market, maintainability, integration, quality assurance— those were the things to worry about. Code bloat? Inefficient use of hardware resources? Not important.

As someone who did a fair amount of assembly-language programming on really small embedded systems in the 1980s, I could never quite swallow that argument. And because of the time I spent writing assembly-language, I deeply appreciate the C language.

C, not C++.

When I want the benefits of object-oriented development, including rich, powerful, well-architected libraries and all the convenient short-hands provided by templates, classes, overloading and inheritance, I use Python or C#. They are truly awesome, powerful, and fast! They produce very compact, portable programs (Python is especially portable, to all those outlying systems Microsoft has no time for...). They save you the trouble of managing memory. They make a lot of hard stuff magnificently easy. And if you don't like them, there are so many others to choose from, like Java, JavaScript, Ruby, Eiffel, and so on and so Forth.

But if I'm trying to write the fastest, tightest code possible, for a driver or server application or library, I believe that C is still the right language: C, not C++. Why?

The thing about C is that it was designed specifically to balance the need for portability against the need in system programming to be able to understand exactly what your code is doing at a hardware level. It was and remains a brilliant abstraction of processor instructions. All those nasty preprocessor directives are an incredibly effective means of setting different contexts so the same code can run on diverse systems. The language itself is pleasantly terse (minimal typing), which is less important these days when our IDEs support code completion, but I'll tell you, it used to matter a lot, and it's elegant! My main point, though, is that C hits the perfect level of abstraction for system code. It's high enough level that you can read a lot of it fast, and it's low enough level that you can find out exactly what is happening at every step.

Why should you know what's happening at every step? Isn't encapsulation a beautiful thing? Not for system code. If you can afford not to care what your components are doing, use a really powerful scripting language. That's often a good choice for mobile devices, too, because byte-code is a lot more compact than native code and is easily portable.

Now, the C++ aficionados will say, perhaps rightly, that for very large projects with multiple teams, C++ can facilitate inter-team communication. It's more type-safe. You can spell out your contracts more clearly using classes and interfaces.

Well, you can do the same things in C, of course, just by taking care. Most of the shortcuts and short-hands of C++ are just that, and the simplicity of C makes you express your APIs in a very concrete way. C forces you to expose what you're doing, where one of the objectives of a lot of C++ code is to obscure (encapsulate) it. And look, if you're thinking about performance, you have to focus on two things:

- Am I doing things the most effective way (am I using the right algorithms)?
- Am I doing things as efficiently as possible (am I avoiding unnecessary steps in my implementation)?

The short-hands and shortcuts of C++ generally make both these factors harder to evaluate.

Also, transparency is particularly valuable in the open-source context, as opposed to the standard corporate model where source-code is kept secret. Even on very large system projects, I believe that the simple clarity of C code makes it the best choice. For instance, say what you will about Linus Torvald's personality, he's a good architect, and Linux remains clean and well factored. It would not be any easier to understand if it were written in C++, and it would likely be much messier under the hood.

Incidentally, I adore Linus' famous rant on this subject in 2007. What a jerk, but he's so right! And he is not alone in having problems with C++. Among its detractors are the likes of Niklaus Wirth ("C++ is an insult to the human brain"), Alan Kay ("I invented the term Object-Oriented, and I can tell you I did not have C++ in mind"), Bertrand Meyer ("There are only two things wrong with C++: the initial concept and the implementation," or "C++ is the only current language making COBOL look good"), and even Donald Knuth ("Whenever the C++ language designers had two competing ideas as to how they should solve some problem, they said, 'OK, we'll do them both'. So the language is too baroque for my taste").

Now, I know C++ programmers will passionately disagree. Bjarne is every bit as arrogant as Linus, and Microsoft is busy drumming up business for a "modern" C++. I'm just speaking my own experience: object-orientation is good when I don't have to care about what's happening at the machine level. If I have to care, C is the language I choose. Still.