The schism, or why C and C++ are different

Ben Klemens

6 May 2006

Those of you who actually read my posts about efficient computing, rather than just going to read the comics at the first sight of the word 'computing', may by now have noticed a few patterns.

The most basic is that standards are important. I know this sounds obvious to you, but if it's so obvious, why do people get it wrong so darn often. Why are people constantly modifying and violating standards that work just fine?

I know many of you have suspected this for a while, but let me state it loud and clear: I am conservative. Rabidly conservative. I think that people need to have a really good reason for not conforming to technical standards, and I think most people don't—they just use the shiniest thing available. A large amount of my writing on technical matters is simply pointing out that well-thought-out technical standards tend to work better than the newest and shiniest, and that the value of stability often more than makes up for inevitable flaws in the standards. Even my work on patents is aimed at making sure that open standards remain open and free to implement.

I originally tried to make this into an essay about both computing standards and general customs, but over the course of writing it, I came to realize that the two are fundamentally different. If somebody doesn't quite conform to your human customs—if they use the wrong fork or speak non-native English or wear ratty t-shirts to the office—then the person will be funny or diverse or annoying or just normal. Meanwhile, if computing standards aren't followed—if somebody gets sick of C's array notation, `array[i][j]`, and decides it looks nicer as `array[i, j]`—then their writing is 100% gibberish and they might as well be speaking Hindu to an English-speaker. Standards-breaking in social settings can be fun; standards-breaking in computing is just breaking things.

So although I usually try to put something in the technical essays that will be interesting to those who could care less about machinery, I don't think any of the below is truly applicable to social norms. Or you can read on and decide for yourself.

[Nor is this a comprehensive essay on standards drift and revolution, because that would take a volume or two. Just file this one as assorted notes on one question with an interesting proposed solution: what to do with all those people who keep trying to revise and update and modify the standards?]

Schisms Intuitively, there's the English-teacher approach to retaining a standard, where we force everybody to stay in line with the basic standard. When you go home to write your pals, your English teacher instructed you, be sure to use perfect grammar at all times.

But another approach is to let the whippersnappers fork. On the face of it, it may seem contradictory to think that splitting a standard in half would somehow make it purer, but under the right conditions, giving those who want to experiment room to do so can be the best approach.

For any technological realm, you've got one set of people who just want features—lots and lots of features, enough to wallow in like they're a bed of slightly moist hundred dollar bills—and you've got another team that wants fewer moving parts, and takes care to maintain discipline and stick to the existing norms. We can bind the two teams together, in which case they will constantly be fighting over little modifications to the system and neither team will be happy. That's what happens with English. Or you can have the schism.

Allow me to cut and paste from Amazon:

The C Programming Language by Brian W. Kernighan, Dennis M. Ritchie
274 pages
Publisher: Prentice Hall PTR; 2nd edition (March 22, 1988)
Amazon.com Sales Rank, paperback: #4,457
Amazon.com Sales Rank, hardcover: #445,546

First edition 228pp, 1978:
Amazon.com Sales Rank, paperback: #60,113

The C++ Programming Language by Bjarne Stroustrup
911 pages
Publisher: Addison-Wesley Professional; 3rd edition (February 15, 2000)
Amazon.com Sales Rank, paperback: #11,797
Amazon.com Sales Rank, hardcover: #6,215

First edition, 327pp.
Amazon.com Sales Rank, paperback: #1,243,918

Things we conclude: C++ is much more complex than C—274pp v 911pp. C++ keeps evolving: from 1986 to 2000, the book has had three editions, over which it has almost tripled in size. People are still buying the 1978 edition of K&R C because it's still correct; the first edition of Stroustrup is so incompatible with current C++ that people can't give it away. Finally, Prentice-Hall *really* needs to lower the price on the hardcover edition of K&R. I mean, *my book* is selling better than their hardcover, which ain't right.

Meanwhile, C is as stable as can be. Cyndi Lauper has put out seven albums since K&R C came out. The changes from first to 2nd ed. of K&R are pretty small—literally, they're a fine print appendix. And, I contend here, it owes its immense stability to Bjarne Stroustrup. With Bjarne putting out a new version of C++ every few years that frolics along with still more features, Prentice-Hall is free to reprint the same version of the C book without people whinging about how it's missing discussion of mutable virtual object templates. The guys who want simplicity and stability buy K&R and the guys who want niftiness and fun features buy Stroustrup and everybody's happy.

The other technical standard I use heavily is TEX, and I'd been meaning, for the sake of full disclosure, to give a critique of TEX comparable to this here critique of

Word[1] Fortunately, Mr. Nelson Beebe already did it for me, in this (PDF) essay entitled 25 Years of TeX and Metafont[2]. The article alludes to exactly the sort of schism in typesetting as in general programming: you've got the people who are totally ignorant of standards and just want the shiniest new thing, and the people who built a standard system that has been stable for the better part of 25 years. Since he's on the standards-oriented team, he gives many examples of how such stability has led to large-scale projects that have significantly helped humanity.

His discussion of its limitations is interesting because there really are features that need to be added to TEX—notably, better support for non-European languages and easier extensibility. But "TEX is quite possibly the most stable and reliable software product of any substantial complexity that has every been written by a human programmer." (p 15) Changing a code base that hasn't seen a bug in fifteen years is not to be taken lightly, and may never happen. Instead, we can expect to see a schism.

Evolution In that 1986 edition of the C++ book, Bjarne wrote this: "since [two standards] will be used on the same systems by the same people for years, the differences should be either very large or very small to minimize mistakes and confusion." I'm going to call this Bjarne's principle.

When you read about the raging debate between Blu-ray and HD DVD (I'm rooting for the one that isn't an acronym), don't think 'now I have to worry about all my stuff being obsolete'. Thank those guys for distracting attention from DVD, which is a nice, stable format that hasn't changed in a decade, ensuring that your stuff has not become obsolete. People have made haphazard attempts to revise the CD format, but thanks to distractions like the MiniDisc and even DVD, your copy of Cyndi Lauper's first album is still the cutting-edge CD standard (specified in The Red Book, 1980). Attempts to incrementally tweak the CD standard never took off. Remember CD+G? If so, you're the only one.

So this is how conservatives evolve. Not from clean standards to floundering in pits of features, but revolutionary breaks from old clean standards to new clean standards. The feature pits are just distractions.

The process of evolution via incremental fixes directly breaks Bjarne's principle, because you get a stream of similar standards that are easily confused and comingled. Corporate-sponsored standards often suffer this failing (but not always), because setting standards that last for two decades and selling frequent updates are hard to reconcile. One company spent a while there naming its document standards with a year—standard '98, standard 2000, et cetera—which in my book means none of the formats are actually standard.

The only way to evolve while conforming to Bjarne's principle is to is to ride a system until it really doesn't do what you need anymore, and then revolt, building a new one that is clearly distinguished from the old, as we saw with DVD's overthrow of CD because CDs truly can not store movies, or Ω's eventual overthrow of TEX because TEX truly can not typeset Tamil.

The trick is to know when to revolt. When is a new feature so valuable that the old

[1] `http://fluff.info/terrible`

[2] `http://www.tug.org/TUGboat/Articles/tb25-1/beebe-2003keynote.pdf`

system should be abandoned? Many a dissertation has been written on this one, and I ain't gonna answer it here. But for well-thought-out technical standards, it's much later than you think, as demonstrated by the active 25-year old standards above.

Back to C vs C++ I copied Bjarne's principle from the first edition of his C++ book, so it comes as no surprise that in the mid-80s, C++ made an effort to conform to Bjarne's principle. In the present day, it just doesn't, and the confusion lies in thinking that it still does.

Even in the first edition, there are incompatibilities between C and the new C++, but just a page or so in the appendix. The author explicitly states (1st ed., p 5) that he's walking into a world of C programmers and C code everywhere, so retaining compatibility is sensible marketing and efficient.

But all those enthusiastically added features, that puffed the third edition up to nine hundred pages, each breaks a little something in raw C. To give a simple example, I use the variable name `template` a few times, and a user wrote me to tell me that his C++ compiler broke on that, because in C++ `template` is a reserved keyword. Bjarne's principle dies another little death.

On the other side, the ISO added a few features to C a decade ago. The most notable for me is designated initializers; I've written several entries here about how much you can get out of this syntactic tweak. However, C++ has no intention of supporting them. This author[3] feels the rationale paper for not using designated initializers gives "arguments that aren't very convincing", and I'd agree.

The `restrict` keyword, also added to C in 1999, does a lot to get code running faster. The authors of C++ have to date rejected the idea of supporting it. But because it's just optimization advice that can be taken or left, here is a valid rule for the parsing of this keyword: replace all instances of `restrict` with a blank space. With no serious technological reason to exclude `restrict`, we're left with just social and æsthetic reasons, and in the subjective balancing of issues, C compatibility and Bjarne's principle was clearly a low priority.

On a positive note, the last revision of C took a number of ideas from C++, after they'd been tested in C++'s feature pit for a few years, including the in-line comments with // [which I use constantly] and the `inline` keyword [which I never use because the compiler will inline functions for you where appropriate]. But in all cases, the rationale was because these features seemed useful and well-tested, not that adopting them would reduce the distance between the two languages.

All of these examples are to show you that modern C++ has basically thrown out Bjarne's principle. Many people still write "C/C++", thinking of them as the same language, comfortably presuming that a C program will compile in a C++ compiler. But that hasn't been really true for maybe fifteen years now. Better would be to just acknowledge the schism. Let them drift further, because things can only get better once the pair are past confusion-maximizing near-similarity, leaving one well-set in its stability and one free to pursue novelty.

[3] `http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=325`