The Problem with Programming
Bjarne Stroustrup, the inventor of the C++ programming language, defends his legacy and examines what's wrong with most software code.
By Jason Pontin

In the 1980s and 90s, Bjarne Stroustrup designed and implemented the C++ programming language, which popularized object-oriented programming and influenced numerous other programming languages, including Java.

C++ remains the archetypal "high level" computer language (that is, one that preserves the features of natural, human language), and it is still used by millions of programmers. Many of the systems and applications of the PC and Internet eras were written in C++. For all that, the language remains controversial, largely because it is notoriously difficult to learn and use, and also because Stroustrup's design allows developers to make serious programming mistakes in the interest of preserving their freedom.

Stroustrup, for many years a researcher at AT&T Bell Labs, is now a professor of computer science in the Department of Engineering, at Texas A&M University, near Houston.

Technology Review: Why is most software so bad?

Bjarne Stroustrup: Some software is actually pretty good by any standards. Think of the Mars Rovers, Google, and the Human Genome Project. That's quality software! Fifteen years ago, most people, and especially most experts, would have said each of those examples was impossible. Our technological civilization depends on software, so if software had been as bad as its worst reputation, most of us would have been dead by now.
On the other hand, looking at "average" pieces of code can make me cry. The structure is appalling, and the programmers clearly didn't think deeply about correctness, algorithms, data structures, or maintainability. Most people don't actually read code; they just see Internet Explorer or Windows "freeze," have their cell phone drop a call, read the latest newspaper story about viruses, and they shudder.

I think the real problem is that "we" (that is, we software developers) are in a permanent state of emergency, grasping at straws to get our work done. We perform many minor miracles through trial and error, excessive use of brute force, and lots and lots of testing, but--so often--it's not enough.

Software developers have become adept at the difficult art of building reasonably reliable systems out of unreliable parts. The snag is that often we do not know exactly how we did it: a system just "sort of evolved" into something minimally acceptable. Personally, I prefer to know when a system will work, and why it will.

**TR:** How can we fix the mess we are in?

**BS:** In theory, the answer is simple: educate our software developers better, use more-appropriate design methods, and design for flexibility and for the long haul. Reward correct, solid, and safe systems. Punish sloppiness.

In reality, that's impossible. People reward developers who deliver software that is cheap, buggy, and first. That's because people want fancy new gadgets now. They don't want inconvenience, don't want to learn new ways of interacting with their computers, don't want delays in delivery, and don't want to pay extra for quality (unless it's obvious up front--and often not even then). And without real changes in user behavior, software suppliers are unlikely to change.

We can't just stop the world for a decade while we reprogram everything from our coffee machines to our financial systems. On the other hand, just muddling along is expensive, dangerous, and depressing. Significant improvements are needed, and they can only come gradually. They must
One problem is that "academic smokestacks" get in the way: too many people push some area as a panacea. Better design methods can help, better specification techniques can help, better programming languages can help, better testing technologies can help, better operating systems can help, better middle-ware infrastructures can help, better understanding of application domains can help, better understanding of data structures and algorithms can help--and so on. For example, type theory, model-based development, and formal methods can undoubtedly provide significant help in some areas, but pushed as the solution to the exclusion of other approaches, each guarantees failure in large-scale projects. People push what they know and what they have seen work; how could they do otherwise? But few have the technical maturity to balance the demands and the resources.

**TR**: The idea behind C++ was that programmers would work harder in return for more-efficient code. Bell Labs wanted a language that a few really smart people would use to write code that would run on computers like Electronic Switching Systems (ESS) that weren't very fast. Today, there are a lot of software developers and computers are very fast. Does that vitiate the point of C++?

**BS**: C++ wasn't designed specifically for the large switching machines, but for a huge range of applications. Bell Labs was the home of an incredible range of interesting projects spanning every scale and using essentially every kind of computer and operating system. But yes, the average Bell Labs programmer was significantly more able than most people's notion of an "average programmer," and reliability and performance (in that order) were considered significantly more important than in most other places.

Performance is still an issue in many of the applications that I'm interested in: responsiveness of interfaces, start-up and close-down time of applications. Software developers have neutralized the astounding performance of modern computer hardware by adding layer upon layer of overelaborate [software] abstractions. We seem to have hit the limits of linear speedup for hardware, but in many cases, we could win a couple of orders of magnitude back from the software.
That said, C++ has indeed become too "expert friendly" at a time where the degree of effective formal education of the average software developer has declined. However, the solution is not to dumb down the programming languages but to use a variety of programming languages and educate more experts. There has to be languages for those experts to use--and C++ is one of those languages.

**TR:** In retrospect, in designing C++, wasn't your decision to trade off programmer efficiency, security, and software reliability for run time performance a fundamental mistake?

**BS:** Well, I don't think I made such a trade-off. I want elegant and efficient code. Sometimes I get it. These dichotomies (between efficiency versus correctness, efficiency versus programmer time, efficiency versus high-level, et cetera.) are bogus.

What I did do was to design C++ as first of all a systems programming language: I wanted to be able to write device drivers, embedded systems, and other code that needed to use hardware directly. Next, I wanted C++ to be a good language for designing tools. That required flexibility and performance, but also the ability to express elegant interfaces. My view was that to do higher-level stuff, to build complete applications, you first needed to buy, build, or borrow libraries providing appropriate abstractions. Often, when people have trouble with C++, the real problem is that they don't have appropriate libraries--or that they can't find the libraries that are available.

Other languages have tried to more directly support high-level applications. That works, but often that support comes at the cost of specialization. Personally, I wouldn't design a tool that could do only what I wanted--I aim for generality.

**TR:** How do you account for the fact that C++ is both widely criticized and resented by many programmers but at the same time very broadly used? Why is it so successful?
BS: The glib answer is, There are just two kinds of languages: the ones everybody complains about and the ones nobody uses.

There are more useful systems developed in languages deemed awful than in languages praised for being beautiful--many more. The purpose of a programming language is to help build good systems, where "good" can be defined in many ways. My brief definition is, correct, maintainable, and adequately fast. Aesthetics matter, but first and foremost a language must be useful; it must allow real-world programmers to express real-world ideas succinctly and affordably.

The main reason for C++'s success is simply that it meets its limited design aims: it can express a huge range of ideas directly and efficiently. C++ was not designed to do just one thing really well or to prevent people doing things considered "bad." Instead, I concentrated on generality and performance.

I'm sure that for every programmer that dislikes C++, there is one who likes it. However, a friend of mine went to a conference where the keynote speaker asked the audience to indicate by show of hands, one, how many people disliked C++, and two, how many people had written a C++ program. There were twice as many people in the first group than the second. Expressing dislike of something you don't know is usually known as prejudice. Also, complainers are always louder and more certain than proponents--reasonable people acknowledge flaws. I think I know more about the problems with C++ than just about anyone, but I also know how to avoid them and how to use C++'s strengths.

And then, of course, you don't expect proponents of languages that lost out in competition with C++ to be polite about it. Software development doesn't have that degree of professionalism--though I hope it eventually will. Science is different in this respect: when a new tool, technique, or theory wins out, people see that as progress. In software, contributions by competitors and predecessors are not widely acknowledged, appreciated, or even understood.

TR: In The Design and Evolution of C++, you claim that Kierkegaard was an influence on your conception of the language. Is this a joke?
BS: A bit pretentious, maybe, but not a joke. A lot of thinking about software development is focused on the group, the team, the company. This is often done to the point where the individual is completely submerged in corporate "culture" with no outlet for unique talents and skills. Corporate practices can be directly hostile to individuals with exceptional skills and initiative in technical matters. I consider such management of technical people cruel and wasteful. Kierkegaard was a strong proponent for the individual against "the crowd" and has some serious discussion of the importance of aesthetics and ethical behavior. I couldn't point to a specific language feature and say, "See, there's the influence of the nineteenth-century philosopher," but he is one of the roots of my reluctance to eliminate "expert level" features, to abolish "misuses," and to limit features to support only uses that I know to be useful. I'm not particularly fond of Kierkegaard's religious philosophy, though.

TR: What do you regret the most?

BS: No regrets! Well, of course I dream of what I might have done differently and better, but seriously, who am I to second-guess, say, 1984 vintage Bjarne? He may have been less experienced than I, but he was no less smart, probably smarter, and he had a better understanding of the word of 1984 than I have. C++ has been used to build many systems that enhance our lives, and it has been a significant positive influence on later languages and systems. That's something to be proud of.

Copyright Technology Review 2006.