December 1, 2006

QUESTIONS FOR BJARNE STROUSTRUP (from Jason Pontin from MIT Technical Review):

Q: Why is most software so bad? Or, if you prefer: Why is software not better than it is? Or, even: Why isn't software as good as it needs to be?

➔ Hmm, some software is actually pretty good by any standards – think of the mars rovers, Google, and the human genome project. That's quality software! (admittedly supported by some quite impressive hardware). Most people – especially experts – would have deemed each of those examples impossible fifteen years ago. Our technological civilization critically depends on software, so if software had been as bad as its 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, the programmers clearly didn't think deeply about correctness, algorithms, data structures, and maintainability. However, most people don't actually read code, they just see IE "freeze", have their cell-phone drop a call, read the latest newspaper story about viruses, and shudder.

The only thing that seems to grow faster than hardware performance is human expectation. Hardware simply cannot keep up, and more and more tasks are pushed into software. At the same time, humans delegate more and more work to computers. The result is a serious overload of individuals and organizations producing software. I think that the real problem is that "we" (that is, the software developers) are in permanent emergency mode and grasping for straws to get our work done. We perform many minor miracles through trial and error, excessive use of brute force, and lots and lost of testing, but – oh, so often – it's not enough.

"We" have become adept at the difficult art of building reasonably reliable systems out of unreliable parts. The snag is that sometimes "reasonably reliable" isn't good enough and 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.

Q: How can we fix the mess we are in?

➔ 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 essentially impossible. People want new fancy gadgets right now and reward people who deliver

them cheaply, buggy, and first. People 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). Even major security problems and crashes seem not to make people willing to change their use of software – and without changes in user behavior, the software suppliers are unlikely to change.

Realistically, we need to heed both "theory" and real-world constraints. We can't just stop to world for a decade while we re-program everything from our coffee machines to our financial systems. For starters, imagine we did that (and managed not to regress into the Stone Age in the meantime), how would we know that our newly designed and built systems would be better than our current mess? On the other hand, just muddling along is expensive, dangerous, and depressing.

Significant improvements are needed and they can only come gradually. They must come on a broad front; no individual change is sufficient. 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, etc. 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 overview and technical maturity to balance the demands and the resources. Worse, reward structures often encourage narrow-view approaches.

It may be useful to point out that I don't exclude my own fields of work from this: no programming language is or could be a panacea, not even C++.

So, looking at my own narrow field, what can we do better? First, there seem to be two dominant schools of thought: "the choice of language doesn't matter" and "choosing the right language will solve your problem". Both are throwing away something essential. My view is that a good language can be a major asset, but only when used appropriately and well. A language doesn't solve problems; it merely helps the expression of solutions.

Object-oriented programming – as pioneered by Simula – can be a major asset. My favorite variant of OO combines it with a serious amount of static (compile-time) type-checking for a more explicit design, earlier error detection, and better performance. In addition, I'm a great fan of generic programming, especially in the areas of data structures and algorithms where GP's more regular approach to design shines. Combining the two can lead to really beautiful, understandable, and efficient code.

However, all the bright ideas in the world won't help us if software development is dominated by half-educated amateurs. We must improve computer science education and

its related fields, and attract more and better students. Only then can we improve the practice of software development, and that only if enough academic efforts stay close enough to practical problems to remain helpful.

Q: C++ was designed for engineers at Bell who were creating code to run on ESS machines. Your idea was that programmers would work harder in return for more efficient code. Bell needed a language that a few people would use to write code that would run all over the world, and computers weren't very fast. Today, the reverse is the case, of course. Does that diminish C++'s usefulness?

➔ Actually, 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 project 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.

Also, performance is still an issue in many of the applications that I'm interested in: responsiveness of interfaces (why do I see character echo delays in Word on my 2GHz, 2Gb computer? I didn't see that on my editor on a shared 1MHz, 1Mb PDP11 25 years ago), startup and close down time of applications, throughput of servers, hard-real time response of "gadgets", scientific computations, etc. "We" have neutralized the astounding performance of modern computer hardware by adding layer upon layer of over-elaborate (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.

Q: Put another way: Is C++ too hard for most programmers?

➔ It shouldn't be. C++ has become "expert friendly," partly by design and partly because the notion of professionalism hasn't taken root in the software development world.

The best uses of C++ involve deliberate design. You design classes to represent the notions of your application, you organize those classes into hierarchies, you express your

algorithms precisely and abstractly (no, that “and” is not a mistake), you use libraries, you build libraries, you devise error handling and resource management strategies and express them in code. The result can be beautiful, efficient, maintainable, etc. However, it’s not just sitting down and writing a series of statements directly manipulating characters, integers, and floating point numbers. Nor is it simply plugging a few pieces of code into a huge corporate framework.

The demands of using C++ well are mainly maturity of system designers and understanding of design and programming technique. They are not primarily demands on the understanding of obscure programming language features. In that sense, the critics are right: Not everybody should write C++, but then I never did claim they should. I do not know if too many people use C++ today, but many do, many more than in the mid-90s, and the use of C++ is still growing in some areas. Obviously, C++ wasn’t too difficult for quite a few people.

C++ is designed to allow you to express ideas, but if you don’t have ideas or don’t have any clue about how to express them, C++ doesn’t offer much help. In that, C++ is not that different from other languages, but it is different from languages/systems/tools/frameworks designed to make it easy to express specific things in a specific domain. The relatively small size of the C++ standard library – primarily reflecting the lack of resources in the ISO C++ standards committee – compared with the huge corporate libraries can be a real disadvantage for C++ developers compared to users of proprietary languages. Given that C++ is designed as a language for writing and using libraries, too many C++ programmers use too few libraries and reinvent the wheel too often. It is far easier to use a good library (such as the C++ STL) than to design and implement it.

Q: In retrospect, wasn't the decision to trade off programmer efficiency, security, and software reliability in exchange for runtime performance a fundamental mistake?

➔ Well, I don’t think I made such a tradeoff. I want elegant and efficient code. Sometimes I get it. The efficiency vs. correctness, efficiency vs. programmer time, efficiency vs. high level, etc. dichotomies are largely bogus.

What I did do was to design C++ as a systems programming language first of all: I wanted to be able to write device drivers, embedded systems, and other code that needed to use hardware directly. I still want that. 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 available – or that they can’t find the libraries that are available.

Other languages have tried to more directly support high-level applications.
That (also) 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.

Q: Put another way: You spent a lot of time avoiding the need to have a C++ runtime, and that made the language really hard to use. But in the end then you got a runtime anyway. What's up with that?

➔ I started out minimal runtime support: just a page of code to allocate and deallocate memory and a few instructions to get the first function started. Compare to that, C++ does indeed have a significant runtime to handle exceptions and its very simple run-time type information mechanism. However, that's still minute compared with just about anything else and can be turned off where needed (e.g., for hard-real-time code).

Maybe you meant automatic garbage collection? You can use that with C++ if you like. That has been a practical alternative for about 15 years now, mostly thanks to the pioneering work of Hans Boehm. It works rather well and will be officially supported in C++0x. Until then, you can download one of the free or commercial C++ garbage collectors. However, you don't have to use a garbage collector. For many applications, it is easier just not to create garbage and the standard library doesn't leak. Often relying on a general resource management strategy is a better idea: Memory isn't the only resource we need to manage: files, locks, network connections, threads, etc. can't be effectively handled except by design.

Q: How do you account for the fact that C++ is both widely criticized and resented by many programmers, but is at the same time very broadly used? Why is it so successful?

➔ The glib answer is "there are just two kinds of languages: the ones everybody complain about and the ones nobody use." That's the polite version. I hear that the less polite version is now being distributed on T-shirts in Cambridge, Mass. (by Quantum books), but they haven't yet hit College Station.

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. People tend to forget this and argue over programming

style or language features using small contrived examples. A programming language is a small – but often highly visible – piece in a huge puzzle.

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. In C++, you can write code that is simultaneously elegant and efficient. Naturally, you can also write code that is neither, and many people couldn't recognize elegance if it walked up and punched them in the nose – but that's true for every language. C++'s strengths lies in the huge range of what it is pretty good at rather than in what it's superb at.

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 (1) how many people disliked C++ and (2) 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 problems with C++ than just about anyone, but I also know how to avoid them and how to use C++'s strengths.

It's also a real problem that many books (even otherwise respectable comparative programming languages books), articles, and courses present an antiquated view of C++. Such as, "C++ is a dialect of C with some OO features bolted on". That hasn't been the case since the early 1980s and even then I'll object to "bolted on". In particular, many popular negative views simply miss the whole area of generic programming and its integration with object-oriented programming and classical data abstraction techniques.

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.

Q: What makes for a good language? (You could talk a little here about the importance of syntax versus semantics, in a general way that will be interesting and comprehensible to a non-programming audience.)

➔ Anything that helps people express their ideas makes a language better. However, people differ in what they want to express and also in how they best express them. Different people really do think differently and their preferences in programming languages and tools partly reflect that. For example, functional programming languages

provide us with many facilities and techniques for elegant expression of ideas, but they have spectacularly failed to directly impact mainstream programming despite being pushed by large parts of the academic establishment for decades. My conjecture is simply that most programmers are uncomfortable with the kind of thinking that is required to use those techniques well. A good language matches the way its users think.

A programming language also influences the way programmers think, but there seem to be distinct limitation to a language's positive influence on its users. Making changes to the way large groups of programmers think is something that takes many years. It was observed decades ago, that you can write Fortran in any language. You can also write C in any language, etc. I'm sure that any widely used language is largely misused in this way. You can write bad code – dangerous code, even – in any language, and people do.

When people compare programming languages, issues of syntax and issues of expressing fairly trivial examples play a huge role. I spend a huge amount of work on such problems under Draconian constraints of compatibility and highly divergent notions of taste in the community. However, this is not where the major gains are made. What you really want to do with language design is to change the way people think; change the way people organize their software.

When I started with C++, essentially everybody in the software industry "knew" that object-oriented programming was too slow to use in real systems and too complicated for "real programmers" to learn to use. Maybe that's even putting too positive a spin on the situation. Most simply hadn't heard of classes, class hierarchies, and object-oriented programming. Knowledge was limited to a rather small group of researchers, academics, and very advanced developers. Even in those groups, the groundbreaking work of Dahl and Nygaard on object-oriented programming, object-oriented design, and Simula was widely unknown or unappreciated. When you went to people doing systems programming, it was even more so. C++ made major contributions in these areas – not by inventing the concepts, but by popularizing them and especially by showing that they were affordable and manageable in real-world projects. In 1980, I repeatedly failed to convince even very intelligent and experienced people about the value of classes and virtual functions. By 1990, the problem was that people were overusing such features and demanding more.

However, object-oriented programming isn't the end of the story. In many areas, we can do better still: many parts of the world are not best described in terms of neat hierarchies. As early as 1986, I was working on ways of parameterizing types (classes) and functions with types. Again, I built on earlier work – and had in fact failed to solve the problems as early as 1980. The result was the C++ templates, which still provide more expressiveness for parameterization than anything else in major use. Using templates, Alex Stepanov built an elegant and extremely powerful set of containers and algorithms (the STL) that became part of the C++ standard library. From there, ideas of parameterization and generic programming have influenced essentially every modern developer community.

The use of exceptions is yet another example. I look for language and library additions that can make major changes. The work on "details" is necessary and can be fun, but it is not what motivates me, and it is not where the major gains for the community is.

Stability and compatibility over decades are very important. That's why I have spent so much time on the ISO C++ standards effort. In another couple of years, the next standard (C++0x) will be ready and C++ will be an even more useful language. You need to balance the need for stability with the need for innovation. Stability is often forgotten in debates about language design, but it's essential for real-world use.

Q: How importance is elegance in a computer language?

➔ Elegance is essential, but how do you measure "elegance"? The lowest number of characters to express the solution to the "towers of Hanoi" problem? My view is that we should look for elegance in the applications built, the code written, rather than in the languages themselves. It would be a stretch to call a carpenter's complicated set of tools – many quite dangerous – elegant. The true worth of a craftsman's tools is only appreciated by craftsmen. On the other hand, my dining room table and chairs are elegant – beautiful, really. I deliberately chose an example from everyday things rather than high art. A language should be good at everyday tasks in the hands of good craftsmen, professionals. It matters less whether the language is beautiful in itself and whether it is effective in the hands of rank amateurs or exceptional artists.

That said, it would of course be best if the language itself was a beautiful work of art, but C++ isn't – though I insist that parts are not bad and far better than their reputation in some quarters. One proof of the intrinsic worth of many of the C++ language features that C++ pushed into mainstream use is that they have now been almost universally adopted by newer languages.

Q: "In the Design and Evolution of C++," you claim that Kierkegaard was an influence upon your conception of the language. Is this a joke? If not, how was the Danish philosopher an influence?

➔ 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 19th 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 his religious philosophy, though.

Q: What is you opinion of pair programming? Do you think it's merely being implemented poorly? Or does he think it's a flawed approach that could never reduce software bugs or vulnerabilities?

➔ I don't know. I don't have enough experience with the way the idea works out in real system development. What I do know is that good design and good code require serious discussion and that you don't get that without many pair of eyes on the design and the code. It is really important to get designers to articulate their ideas, so anything that helps that – documentation, tutorials, presentations, discussions – is good. Many opinions, ideas, backgrounds, and skills go into good software. My favorite design tool is a blackboard, and I try not to hog the chalk. Does pair programming help bring ideas out into the open, or does it just result in group think in a tiny group of two?

Q: What do you regret the most?

➔ 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 enhances our lives and it has been a significant positive influence on later languages and systems. That's something to be proud of.

What I want most of all, though, are more and better standard libraries, but those are hard to get because the standards committee is a group of volunteers without a budget.

And C++ is still evolving. Soon, we'll know exactly what C++0x will look like and there is a lot of new and interesting work being done in the C++ world. For example, only this year we (a couple of my students and I) figured out how to design multi-methods (functions dispatched on more than one dynamic type) so that they are about as fast as virtual functions, execute in constant time, obey the usual overload resolution rules, and never throw exceptions. Also, over the last few years we (mostly people from the C++ standards committee from Adobe, Indiana University, and Texas A&M University)

figured out how to combine C++ template's unmatched flexibility and performance with precise specification, leading to better overloading and much better error handling.

I do look back, though, partly to document what happened (there are far too many myths "out there") and partly to try to learn from history. I wrote a long paper on the early history of C++ for the ACM conference of the history of programming languages (available from my publications page) and I'm writing another one for next year's ACM HOPL conference to follow up from where my first HOPL paper left off. My book "The Design and Evolution of C++" documents a lot of the design decisions.

Reflection and honest evaluation of what worked and what did not is essential for progress. Documenting the past is also a great way of acknowledging debts and thanking contributors.

Q: Please name the coolest and lamest programs ever written in C++, and say what worked or didn't work about them.

➔ First, what is a program to most people is a complete system, not code. Second, there is not concept of 100% pure C++. C++ was designed to provide part of systems writing in a variety of languages, so other languages play parts of the systems I mention: Google – can you even remember the world before Google? (it was only about 5 years ago). What I like about it is its performance under severe resource constraints. There are also some really neat parallel and distributed algorithms. The first browsers – can you imagine the world without the web? (it was only about 10 years ago). Other programs that I find "cool" are examples of embedded systems code: the scene analysis and autonomous driving systems of the mars rovers, a fuel injection control for a huge marine engine. There is also some really cool code in Photoshop's image processing and user interfaces. Again, what I like from a code perspective is the way these programs are structured to be reliable and responsive under pretty harsh resource constraints. Some of Photoshop's ways of managing internal complexity (e.g. GUI layout and access from image processing algorithms to the pixel data) are just beautiful.

If you look at the code itself, rather than considering the end product, we could look at something like the shape example, which I borrowed from Simula. It's still the language-technical base of most graphical user interfaces, such as the one on your computer or your iPod or whatever. A more modern example would be the find or the sort algorithm in the STL. That's the language technical basis for much modern high-performance C++ code, especially of programs that need to do simple operations on lots of data. What is common to those to kinds of code is that it cleanly separates concerns in a program, allowing separate developments of parts, simplifies understanding, and eases maintenance. These basic language techniques simply allow separate "things" to be separately represented in code and combined (only) when needed. However, code is something appreciated by programmers, rather than most people. I have always been a bit envious of graphics people, robotics people, etc. What they do is so concrete and visible;

what I do is invariably invisible and incomprehensible to most people. I know many mathematicians feel the same way.

Sorry, I'm not going to shame anyone by naming their work "the lamest C++ program ever." It's oh, so tempting, but no, it wouldn't be fair.

Q: After structured programming (a la Pascal) and object-oriented programming (a la C++), what will be the next big conceptual shift in the design of programming languages? A few years ago, in our TR10 issue, Technology Review put its money on aspect-oriented programming. Does it represent a conceptual shift of the same kind that OO did? Is it catching on?

➔ I hope you didn't put too much money on it ☺. I don't see aspect-oriented programming escaping the "academic ghetto" any day soon, and if/when it does, it will be less pervasive than OO. When it works, aspect-oriented programming is elegant, but it is still uncertain how many applications significantly benefit and how easy it is to integrate the necessary tools into an industrial scale programming environment.

One reason I am cautious is that I saw an early system built on similar ideas, R++, struggle to gain acceptance some 15 years ago. It did well for a major application, but repeatedly genuinely enthusiastic people found only few examples that were genuinely crosscutting in their programs, that they could handle those examples with workarounds, that introducing major changes into their tool chains and processes was complicated and expensive, and that educating new developers was difficult and time consuming. Naturally, aspect-oriented programming may avoid some of these problems, but to succeed it need to dodge all, and more. A major new idea will succeed only if it is sufficiently capable in every relevant area. To succeed on a large scale, you must be good enough in all areas (even some you have never heard of) rather than just superb at one or two things (however important). This is a variant of the simple rule that to function, all essential parts of a machine must work – remove one and the machine stops. The trick is to know which parts are essential. Please note that I'm not saying "aspect-oriented programming doesn't work", but to be "the next big thing" you have to provide major gains in an enormously broad range of application areas.

I don't know what the next major conceptual shift will be, but I bet that it will somehow be related to the management of concurrency. As programmers, we have been notoriously bad at thinking about lots of things happening simultaneously and soon our everyday computers will have 32 cores.

You didn't mention "generic programming." It is definitely worth thinking about in this context. Over the last decade, it has made a major change to the way C++ libraries are designed and has already led to addition of language features in Java and C#. I don't think of generic programming as the "next paradigm" because C++ has directly

supported it since about 1990 and the C++ standard library crucially depends on it. What if the “next big thing” has already arrived and nobody (except programmers) noticed?

I invariably list generic programming as one of the four fundamental styles of programming supported by C++, alongside object-oriented programming. From a language point of view, it is roughly what Christopher Strachey described as parametric polymorphism as opposed to ad hoc polymorphism, which we know as object oriented programming. Generic programming focuses on algorithms and the requirements of algorithms on their arguments. It is therefore most effective when we are dealing with problems that have a clean mathematical formulation. For such problems, generic programming often leads to shorter, cleaner, and faster code than other formulations; it more directly reflects the solution domain and applies to a greater range of data. For most people, generic programming starts with Alex Stepanov’s work on the STL, as adopted by the C++ standard library.

I use “multi-paradigm programming” to describe the current incomplete synthesis of traditional (C-style) programming, data abstraction, object-oriented programming, and generic programming that dominates the most advanced and effective uses of C++. Some would consider what is emerging here a new paradigm. I wonder how this will fit with concurrency?

Please note that I don’t actually believe in one of the popular uses of the word “paradigm” (going back to Kuhn) when applied to programming: I do not believe that a paradigm completely replaces previous paradigms in one revolutionary moment (or “shift”). Instead, each programming paradigm adds to what worked previously, and as a paradigm matures, it is increasingly integrated with previous paradigms. Kristen Nygaard was fond of saying that multiplication didn’t completely replaceaddition, and – by analogy – whatever would come after object-oriented programming would include object-oriented programming as a subset. I agree. The evolution of C++ was guided by this view, and the evolution of Java and C# provides further examples.

Q: Computer languages remain generally difficult to learn. One might argue that, for computers to become more than "helper" tools that enable mass computations and widespread communications (ie, through the internet), they must evolve again - and one key may be in simplifying the process of coding so that more individuals are able to participate in development. The 4GL revolution, in fact, was designed to more closely emulate our understanding of how humans interact with objects in the physical world in an effort to simplify and speed the process of building applications. It worked. So how do we make programming even more accessible to people with progressively less training required?

➔ I think that would be misguided. The idea of programming as a semi-skilled task, ideally practiced by people with a few months’ training, is dangerous. We wouldn’t

tolerate plumbers or accountants that poorly educated. We don't have as an aim that architecture (of buildings) and engineering (of bridges and trains) should become "even more accessible to people with progressively less training required." One serious problem is that currently too many software developers are under-educated and/or under-trained.

Obviously, we don't want our tools – incl. our programming languages – to be more complex than necessary, but one aim – among others – should be to make tools that will serve skilled professionals; not to lower the level of expressiveness to serve people who can hardly understand the problems, let alone express solutions. We can and do build tools that make simple tasks simple for more people, but let's not let "most people" loose on the infrastructure of our technical civilization or force the professionals to use only tools designed for amateurs.

Actually, I don't think it is the programming languages that are so difficult to learn. For example, every first-year university biology textbook contains more details and deeper theory than even an expert-level programming language book. Most applications involve standards, operating systems, libraries, and tools that far exceed modern programming languages in complexity. What is difficult is the appreciation of the underlying techniques and their application to real-world problems. Obviously, most current languages have many parts that are unnecessarily complex (though we can't compete with, say, the complexity of the fruit flies that biologists study), but the degree of those complexities compared to some ideal minimum is often exaggerated. We need relatively complex language to deal with absolutely complex problems. I note that English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful.

Q: Some say that the future lies in collaborative programming - in some cases across multiple locales. This presents challenges for languages - do you believe it's true, and if so, how can C++ evolve to address those challenges?

➔`It has already happened. Most major systems are developed and maintained by geographically distributed groups of people.

Obviously, anything that helps express modularity would be an asset. This is an area that requires much research and especially practical testing. I am not at a point where I would recommend a specific direction of work.

Q: Please talk about the pros and cons of maintaining backward compatibility with the existing knowledge base (for example, consider your determination to maintain high

compatibility to C when you developed C++). It would seem that a clean break might produce leaps of progress; but is this a realistic proposition?

➔ Java shows that a (partial) break from the past – supported by massive corporate backing – can produce something new. C++ shows that a deliberately evolutionary approach can produce something new – even without significant corporate support. To give an idea of scale: I don't know what the marketing budget for Java has been so far, but I have seen individual newspaper advertisements that cost more than the total of AT&T's C++ marketing budget for all time. "Leaps" can be extremely costly. Is such money well spent? Maybe from the point of view of corporate budgets and corporate survival, but given a choice (which of course I'll never have), I'd spend that money on research and development of evolutionary changes. Note that almost by definition research money are used to fund attempted "leaps" that tend to fail while competing with evolutionary changes.

However, "evolution" mustn't be just an excuse for doing things "the way we always did." I would like for evolutionary changes to occur at a faster pace than they do in (say) C++ and I think that's feasible in theory. However, that would require funding of "advanced development," "applied research", and "research into application" on a scale that I don't see today. It would be essential to support the evolution of languages and libraries with tools to ease upgrades of real systems and tools that allowed older applications to survive in environments designed for newer systems. Users must be encouraged to follow the evolutionary path of their languages and systems. Arrogantly damning older code as "legacy" and recommending "just rewrite it" as a strategy simply isn't good enough. "Evolutionary languages" tend to be very conservative in their changes because there is no concept of supporting upgrades. For example, I could imagine accepting radical changes in source code if the change was universally supported by a solid tool for converting from old-style to new-style. In the absence of such tools, language developers must be conservative with the introduction of new features and application developers must be conservative with the use of language features.

One problem with an evolutionary approach is that there are few academic outlets for incremental change, especially not when that incremental change directly relates to real-word systems. This is a sign that computer science and software engineering haven't yet emerged as the theoretically and empirically well-founded foundation of real-world software development. Theory and practice seem rarely to meet and researchers are pushed away from real-world software towards less messy academic topics. Conversely, many developers completely ignore research results.

Another problem is that corporate funding is focused on areas where the managements see potential for corporate product differentiation. Java's success could be seen as an accident in the sense that the basic technology was useful well beyond Sun's user base. To contrast, C++'s success was the result of a deliberate effort to "seed" the community with ideas that would only indirectly benefit AT&T.

Against my case for evolutionary changes must be taken the fact that most people who want compatible changes tend to argument for really minor additions and for full compatibility. This can lead to bloated messes that incur the cost of change, yet reject everything genuinely new. The aim of work on languages, tools, and techniques must be to change the way people think – to aim for major changes in the way people work. In reality, most people are far more comfortable fiddling with minor details.

A major issue in language evolution is control of its direction. For a new language, users – in principle – have a choice: you can adopt it or not (though this becomes less clear cut if the new language is heavily pushed by your platform supplier). Once you use a language, it is hard to ignore new features (e.g., you might not like a feature, but a colleague or a library supplier may think it great and use it) and impossible to ignore incompatible changes. The ISO C++ standards process relies on volunteers, who can devote only part of their time to the standards work. This implies that it is slow moving. It also seems to imply that end-users are consistently underrepresented as compared to suppliers of compilers and tools. Fortunately, the C++ committee has always been able to attract many dozens of active members for its meetings and many more online, so it avoided the parochialism of a small group. Until very recently, academics have been completely absent.

C++ provides a nice and extended case study in the evolutionary approach. For starters, see my "The Design and Evolution of C++" and my papers to the ACM History of Programming Language conferences. C compatibility has been far harder to maintain than I – or anyone else – had expected. Part of the reason is that C has kept evolving, partially guided by people who insist that C++ compatibility is neither necessary nor good for C. Another reason – probably even more important – is that organizations prefer interfaces that are in the C/C++ subset so that they can support both languages with a single effort. This leads to a constant pressure on users not to use the most powerful C++ features and to myths about why they should be used "carefully," "infrequently," or "by experts only." That, combined with backwards-looking teaching of C++, has led to many failures to reap the potential benefits of C++ as a high-level language with powerful abstraction mechanisms.

Q: Elements of the C++ syntax have been picked up and used by a variety of languages - Java of course being the obvious first example. Is this flattering? Or unnerving, given the apparent interest in picking up the syntax but not, say, the core reliability and security features of C++?

➔ "Imitation is the most sincere form of flattery", though I could have hoped for a greater appreciation of some of C++'s underlying principles and especially for a more honest acknowledgement of the antecedence of features.

Q: Would you consider .NET one of your progeny - it does, after all, offer a high level of extrapolation and "pluggable" components. It's also the language most desired in corporate job candidates. What are its pros and cons?

➔ .Net is “the progeny” of a large organization, though Anders Hjelsberg has a large hand in it through C# and its libraries. I suspect that C++ played a significant role, but primarily through MFC (which is not one of the more elegant C++ libraries) and as an example of something perceived as needing major improvement. C# as a language is in some ways closer to C++ than Java is, but the main inspiration for .Net and C# is Java (e.g. J2EE). Maybe C++ should be listed as a grandparent for .Net but as both a parent and a grandparent of C#.

.Net is a huge integrated system backed by Microsoft. That’s its major advantage and disadvantage. If you want to write software to be consumed by others on .Net, you have two basic choices: C# or C++/CLI. If you just want to write an application you can do so using “native code” in any language (e.g. ISO C++ without proprietary extensions).

Personally, I’m a great fan of portability. I want my software to run everywhere it makes sense to run it. I also want to be able to change suppliers of parts of my system if the suppliers are not the best. Obviously, suppliers of huge integrated systems, such as .Net and Java, see things differently. Their claim is that what they provide is worth more to users than independence. Sometimes they are right, and of course some degree of integration is necessary – you cannot write a complete application of any realistic size without introducing some system dependencies. The question is how deeply integrated into the application those system dependencies are. I prefer the application to be designed conceptually in isolation from the underlying system, with an explicitly defined interface to “the outer world” and then integrated through a thin layer of interface code.