The C Family of Languages

Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling

This article appeared in *Java Report*, 5(7), July 2000 and C++ *Report*, 12(7), July/August 2000.

The C family of languages—C, C++, and Java—has dominated commercial programming for over 30 years. Today, all three languages are at a turning point:

- The second ISO/ANSI C standard has just been published (C99 was officially released in December 1999). C continues to be one of the most influential languages in the world, particularly in the field of embedded systems.
- The first official update to the ISO/ANSI C++ standard will be completed in October 2000. C++ is one of the most widely used commercial programming languages in the world, with unparalleled support for both object-oriented and generic programming, and continues to experience steady growth.
- Java popularity continues to grow in various areas, from client-side to server-side programming. Sun has recently decided that Java can best flourish as a de facto standard, instead of as a formal ISO/ANSI or ECMA standard, and has abandoned formal standardization efforts.

What has made the C family of languages so dominant? What can we expect to see in the near future in these and related languages? This landmark "state of the union" interview, published simultaneously this month in both C++ *Report* and *Java Report*, brings together three legends who hardly need any introduction: Dennis Ritchie, Bjarne Stroustrup, and James Gosling, the creators of C, C++, and Java.

Looking Back: Where Have We Come From?

Q: Why has the C family of languages become so successful and so widely used?

DENNIS RITCHIE: This has always been a bit of a mystery to me to understand in any kind of detail. Obviously the use of C[1] was during early times (meaning the '70s and much of the '80s) considerably encouraged by its use as the *lingua franca* of Unix during the period that Unix was growing in the research and academic community, and then when Unix was taken up as the software basis for the workstation industry of the '80s. This in turn had to do in part with the non-political nature of C and Unix (not tied to a power in computer hardware until post-1984). There were also technical and semi-technical aspects: the language turned out to be well-placed both for describing things at a high enough level so that portability across hardware was feasible, but simple enough in its requirements to make it cheap to implement.

BJARNE STROUSTRUP: C and C++ became popular because they were flexible, cheap, and more efficient than alternatives. C owes much of its initial popularity to the popularity of Unix. C++ owes much of its initial popularity to its high degree of compatibility with C.

It was very important success of C and C++ that AT&T didn't try to monopolize these languages, but allowed its researchers to support the creation of alternative implementations. Also, AT&T fully supported ANSI and ISO standardization of C and C++

[1] B. Kernighan and D. Ritchie. **The C Programming Language, 2nd edition** (Prentice Hall, 1998) ISBN 0131103709.

as soon as these efforts started. There was no systematic marketing of C or C++ before they became established languages and multiple vendors started competing. This non-commercial spread of C and C++ appealed strongly to many programmers.

Java is a very different design from the other two languages and appears to have a very different philosophy. It owes much of its initial popularity to the most intense marketing campaign ever mounted for a programming language. From its initial commercial debut onward, Java was marketed as radically different from, and better than, all other languages. Interestingly, Java was marketed to individuals at all organizational levels—not just to programmers.

I suspect that the root of many of the differences between C/C++ and Java is that AT&T is primarily a user (a consumer) of computers, languages, and tools, whereas Sun is primarily a vendor of such things.

Just to remind people: Both C and C++ were invented in the Computer Science Research Center of Bell Labs in Murray Hill and found their initial serious use within Bell Labs and AT&T. Then, Bell labs was the R&D arm of AT&T. Now, part of Bell Labs is the R&D arm of Lucent and part stayed with AT&T under the name "AT&T Labs."

None of these languages was radically different or dramatically better than other contemporary languages. They were, however, good enough and the beneficiaries of luck and "social" factors such as Unix, low price, marketing (Java only), etc.

Among technical factors, C and C++ benefited from their closeness to machine and absence of artificial restrictions on what can be expressed. That allows low-level systems work to be done in these languages and for the full performance of a machine to be delivered to its users. Java benefited from running in its own virtual machine and from coming with a large set of libraries that decrease the time needed for a programmer to become productive. Unix gave a similar boost to C. In contrast, the C++ world suffers from fragmentation of its huge base of libraries, many of which are proprietary and supplied by competing vendors.

James Gosling: I think that the number one reason is that it's been generally a very pragmatic family of languages. By and large they weren't experiments in language design; they were put together as tools by people who wanted to do something else. C was very largely driven by the writing of the Unix operating system and all the utilities in it, and so a lot of the things that are in C are straight from what it takes to build an efficient operating system, and also what it takes to do that on a machine that only has 32K.

Q: What were your major original design goals for [C/C++/Java]? What problems was the language meant to solve? What problems was the language not meant to solve?

Ritchie: The point of C (as distinct from its immediate predecessor B) was to take a language that was designed with word-oriented machines in mind and adapt it to the newer hardware that became available, specifically the PDP-11. It did not take long to learn that the kind of things introduced early on would also make it adaptable to much different machines.

Stroustrup: My initial aim for C++ was a language where I could write programs that were as elegant as Simula programs, yet as efficient as C programs.

The projects I was considering at the time had to do with distributing operating systems functions and applications across a number of computers in a local-area network. I had to consider both the complexity of the total system and the efficiency of the lowest-level facilities. The object-oriented facilities from Simula helped with the former, and the systems-programming facilities of C with the latter.

Gosling: Like C, Java's goals were really around being able to build relatively specific kinds of software, and we were really going for very distributed, very reliable,

interacting-with-people kinds of software—ones where things like reliability mattered a huge amount, things like security mattered a huge amount, and they all mattered enough that there was a willingness to take a performance hit for them.

For example, one of the differences between Java and C is that Java has real arrays and the arrays get bounds-checked, and that turns out to be important for both reliability and security. If you look at the history of security breaches on the Internet, some really large fraction of them are exploiting buffer overflows, statically allocated arrays that people just go over the end of. If you look at logs of bug-fixing that people go through, it ends up being that memory integrity issues are a huge fraction of where the problems come from.

And yet, if you try to do something like what Java does, where it's really strict about the memory model, there is some performance penalty. It turns out that with reasonably good optimizing compilers the performance penalty can become almost unmeasurably small, but it does require a high-quality compiler to do that, and 20 years ago nobody was building compilers that did the kind of optimization that compilers do today. The goals about security and reliability led to differences that caused performance problems that could only really be recovered with high-quality compilers, and a lot of the original C design was motivated by "what can you get good quality code out of without some really magnificently optimizing compiler."

Q: How have those goals for [C/C++/Java] changed over time, if at all? Why?

RITCHIE: My own goals for C haven't changed much in many years, and I haven't been a central player in the changes in either the 1989 or 1999 standards. The 1989 ANSI and ISO standard codified things considerably better than our original document; the most important introduction was something that I should have done earlier on (function prototypes). By the way, Bjarne's earliest work on what became C++ was probably the most important direct influence on this.

STROUSTRUP: I don't think my overall goals for C++ have changed much. I still want to write programs that are simultaneously elegant and efficient. I still want serious systems programming—including work at the machine level and on resource-constrained systems—to be feasible in C++.

The major shift in emphasis/style comes from finally having templates and exceptions available. From the first paper I wrote on C++'s ancestor, C with Classes, I knew that I needed an efficient and statically checked way of accessing containers. Templates provided that, and the STL provided a set of techniques for practical use of such containers.

A more general answer: I tried to document my aims and design considerations for C++ in *The Design and Evolution of* C++.[2] That is the place to go for questions about why C++ is the way it is and why it didn't take some other plausible path.

GOSLING: I guess if anything the goals for scalability and reliability have just gotten stronger and stronger, because they've turned out to pay off in things like development time and people value that kind of thing a lot. It was kind of an odd position to take, to try to build a language that was kind of in this space between interpreted languages, where everything is flexible and everything is dynamic, and static languages like C, where everything is statically precompiled. Java is kind of in this netherworld in between them, and given the way the world has evolved to become even more networked than before, even more conscious of security issues, even more in contact with human beings at every moment, the basic goals have pretty much stayed there. They've just gotten, if anything, stronger.

[2] B. Stroustrup. **The Design and Evolution of C++** (Addison-Wesley, 1994) ISBN 0201543303.

Q: As you designed [C/C++/Java], what were the hardest features to get right, or the hardest features to create in a way that your initial users would accept?

RITCHIE: Generally, there is the question of what to put in and what to leave out. Probably the oddest aspect of C compared with other languages outside its immediate family is the declaration syntax, in which (in a way coming from Fortran) a type is mentioned, then variables decorated in a way that reflects their use in expressions. There are many who don't like this, so whether I got it right is an open question.

A closely related aspect is the treatment of arrays and pointers. Given the perenniality of questions about this and the contortions that C++ goes [through] to move away from C on this, it might well be counted as a mistake.

STROUSTRUP: Getting the right degree of C compatibility has been a constant struggle. For what I wanted to do with C++, I needed stronger static checking than C offered. However, every incompatibility is an inconvenience to somebody—especially to somebody who value old code (however badly written) over new facilities (however useful). Programmers, and marketeers, can be very ruthless in their condemnations when old code might be broken—even if that code is written in another language or in a non-standard dialect.

However, I consider the hassle worth it because the C and C++ communities benefit from a genuine degree of compatibility. Also, much of my work on the non-abstraction areas of C++ has been fed back into Standard C. Examples are function declarations/prototypes, const (Dennis also had a hand in their design), declarations wherever statements can appear, and the // comments.

Templates took a long time to get right and reasonably complete. In a sense, their design started with my (largely failed) experiments with macros in "C with Classes." It is easy to design a parameterized type facility that serves the basic need of getting containers statically type safe. To deliver the flexibility and efficiency that I consider essential is far more difficult and requires experimentation and feedback from real use. Generic programming, as currently practiced in Standard C++, goes beyond what most people considered possible (especially in a language that aims at uncompromising efficiency). My guess is that this is an area where we will see significant progress in the next few years—and where language facilities might be needed to support new techniques.

GOSLING: There were a lot of really hard design issues, and they were all over the map. The ones that people really cared about, for instance, included this notion that Java has this thing called an interface. A lot of the design of interfaces was borrowed pretty straightforwardly from Objective C, and it's got some pluses and minuses and it was a real tough balancing act.

Q: Did you ever add features that your users didn't appreciate as much as you did, and then have to deprecate or remove them later? What did you learn from the experience?

RITCHIE: I added some things under some pressure from users that I don't think were done well. Enumeration types are a bit odd, bitfields are odder. The "static" keyword is very strange, expressing both a storage lifetime and what the standard calls "linkage" (external visibility). There are many odd bits here and there.

I can't remember features I added that I had to remove except for some tiny bits, like the "entry" keyword.

STROUSTRUP: When I designed C with Classes and C++, I was very keen on the idea that a class defined the environment in which the code of its member functions operate (I still am). This led to the notion of constructors that establish that environment (invariant) and acquire the resources needed. Destructors reverse that process (releasing resources). These ideas are at the root of the design of exceptions and resource management. For

example, see the new Appendix E: "Standard-Library Exception Safety" of *The* C++ *Programming Language*.[3] That appendix can be downloaded from my home pages.[4]

Based on that line of thinking, C with Classes also allowed you to define a call() function that was called before entry into a member function and a return() function that was called after exit from a member function. The call()/return() mechanism was meant to allow a programmer to manage resources needed for an individual member function invocation. This allowed me to implement monitors for the first task library. A task's call() grabbed a lock and its matching return() released the lock. The idea came partly from Lisp's :before and :after methods. However, the notion wasn't perfectly general—for example, you couldn't access arguments for a member function call from call(). Worse, I completely failed to convince people of the usefulness of this notion, so I removed call() and return() when I defined C++.

I recently revisited this issue and I wrote a template—in Standard C++—that wraps calls to an object by arbitrary prefix and suffix. A paper describing this, *Wrapping* C++ *Member Function Calls*, appeared in last month's issue of the C++ *Report*.[5]

I tried to do something about the declarator syntax. I considered using postfix `->` as an alternative to postfix * and experimented with having the name appear in alternative places in the declarator sequence. For example:

```
int (*p)[10]   // pointer to array of int
int p->[10]    // alternative
int->[10] p;   // alternative
->[10]int p;   // alternative
p: ->[10]int;  // alternative
```

Unfortunately, I fumbled that set of ideas, and nothing came of it. The few examples of improvements to the basic C syntax that I implemented locally received a reception that caused me to back them out with days.

GOSLING: No, I had this personal rule that by and large I didn't put anything in just because I thought it was cool. Because I had a user community the whole time, I'd wait until several people ganged up on me before I'd stick anything in. If somebody said, "Oh, wouldn't that be cool," by and large I ignored them until two or three people came up to me and said "Java ought to do this." Then I'd start going, well, maybe people will actually use it.

There's this principle about moving, when you move from one apartment to another apartment. An interesting experiment is to pack up your apartment and put everything in boxes, then move into the next apartment and not unpack anything until you need it. So you're making your first meal, and you're pulling something out of a box. Then after a month or so you've used that to pretty much figure out what things in your life you actually need, and then you take the rest of the stuff—forget how much you like it or how cool it is—and you just throw it away. It's amazing how that simplifies your life, and you can use that principle in all kinds of design issues: not do things just because they're cool or just because they're interesting.

Q: If you could go back in time, knowing what you know now, what might you do differently in designing [C/C++/Java]? Why?

RITCHIE: Evidently there were no gigantic mistakes. There are of course many aspects I'd at least rethink (some detailed above).

---

[3] B. Stroustrup. **The C++ Programming Language, Special Edition** (Addison-Wesley, 2000) ISBN 0201700735.

[4] http://www.research.att.com/ bs.

[5] B. Stroustrup. **"Wrapping C++ Member Function Calls"** (C++ *Report*, 12(6), June 2000).

STROUSTRUP: You can never go back, and doing something over again is pointless.

As I have often said, I consider not shipping a larger standard library my biggest mistake. In addition, I clearly wish I could have done something about the C declarator syntax. Having had templates integral to the language early on would have helped many use C++ better. However, I did not at the time think I knew how to do those things well enough. My reluctance to bully my way through have saved me—and my users—often enough for me to be reluctant to second guess my younger self in these matters.

Just now, I'm very happy to finally have many implementations available that approximate the standard. I can now write code that really uses Standard C++ as it was meant to be used—and move elegant code from implementation to implementation and from machine to machine. The fun I'm having programming and the quality of the code I can now write keep me from dwelling on possible past mistakes.

GOSLING: There are a bunch of things that I'd do differently. There are a number of things that I'm not entirely happy with and it's not clear what the right answer is. I'm not really happy with the schism between interfaces and classes; in many ways it feels like the right solution wouldn't get in the way.

There have been a number of things—like, for example, multiple return values—that these days I kind of wish I had added. That was one where I really liked the feature and most other people just sort of went, "Huh?" Another one that sort of went that way was that I had been going down this route of having a bunch of stuff to do with preconditions and postconditions and assertions in an Eiffel-like way, and actually in the community of people who were using it at the time the average answer was, "Huh? Why would I ever want that stuff?" I think that the average developer today would say that too. But then there's a pretty reasonable crowd of people who believe in things like Design By Contract, and it's one of these things where a lot of people feel like, "If only the rest of the world was educated enough to understand what this is about, they'd be better off." And I actually kind of agree with that. The problem is that most of the world could actually care less.

There are some things that I kind of feel torn about, like operator overloading. I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++. I've spent a lot of time in the past five to six years surveying people about operator overloading and it's really fascinating, because you get the community broken into three pieces: Probably about 20 to 30 percent of the population think of operator overloading as the spawn of the devil; somebody has done something with operator overloading that has just really ticked them off, because they've used like + for list insertion and it makes life really, really confusing. A lot of that problem stems from the fact that there are only about half a dozen operators you can sensibly overload, and yet there are thousands or millions of operators that people would like to define—so you have to pick, and often the choices conflict with your sense of intuition. Then there's a community of about 10 percent that have actually used operator overloading appropriately and who really care about it, and for whom it's actually really important; this is almost exclusively people who do numerical work, where the notation is very important to appealing to people's intuition, because they come into it with an intuition about what the + means, and the ability to say "$a + b$" where a and b are complex numbers or matrices or something really does make sense. You get kind of shaky when you get to things like multiply because there are actually multiple kinds of multiplication operators—there's vector product, and dot product, which are fundamentally very different. And yet there's only one operator, so what do you do? And there's no operator for square-root. Those two camps are the poles, and then there's this mush in the middle of 60-odd percent who really couldn't care much either way. The camp of people that think that operator overloading is a bad idea has been, simply from my informal statistical sampling, significantly larger and certainly more vocal than the numerical guys.

So, given the way that things have gone today where some features in the language are voted on by the community—it's not just like some little standards committee, it really is large-scale—it would be pretty hard to get operator overloading in. And yet it leaves this one community of fairly important folks kind of totally shut out. It's a flavor of the tragedy of the commons problem.

Using the Languages

Q: In your experience, what are the most common mistakes developers make in [C/C++/Java]?

RITCHIE: I don't really know the answer to this question. I would suppose that the low-level mistakes have to do with type errors, and especially with array subscript and pointer references. Although the rules about what you can do are pretty clear, implementations traditionally do little checking..

I would also venture that the really big problems have to do with not thinking out the large-scale structure of big systems: what's visible where, naming of things, who can change what. C itself does very little to help you here; you have to design a scheme.

STROUSTRUP: Using C++ just as if it was C or Smalltalk. That leads to seriously suboptimal C++. To use C++ well, you have to adopt some native C++ styles. These tend to be distinguished by small concrete classes and templates. A programming style that is overly influenced by C tends to use a lot of arrays, clever pointer manipulation, casts, and macros rather than standard library facilities (such as vectors, strings, and maps). A programming style that is overly influenced by Smalltalk tries to cram every class into a hierarchy and to overuse casts (and often macros). In either case, key abstractions tend to disappear in a mess of implementation details and people get themselves (unnecessarily) tied in knots with allocation, pointers, casts, and macros.

One common mistake that particularly annoys me is the use of complicated base classes with lots of data members as part of interfaces offered to application builders (users). Abstract base classes make much better interfaces to services. I have tried to root out such misuses for well over ten years: I added abstract classes in 1989 after failing to get the idea across without the support of an explicit language feature.

The idea is really simple:

```
class interface {            // seen by users
            // pure virtual functions
};

class my_implementation : public interface {
            // data
            // functions
            // overriding functions
};
```

When user code sees only "interface," it is immune to changes to "my_implementation."

If common data structures or operations are needed for implementations they can be added in a base class that is visible only to implementers:

```
class common {            // seen by implementers
            // data
            // functions
};

class my_implementation : public interface, protected common {
```

```
            // data
            // functions
            // overriding functions
};

class your_implementation : public interface, protected common {
            // data
            // functions
            // overriding functions
};
```

This is one of the simplest and most fundamental uses of multiple inheritance.

What many people need to get *much* more out of C++ is not new features, it is simply a more appropriate design and programming style. I have written several papers on this (see my "papers" page among my home pages). In particular, *Learning Standard C++ as a New Language*[6] compares simple exercises done in C style and C++ style.

GOSLING: Probably the most pervasive mistake is not doing a tasteful job of object-oriented programming. There are these people who don't understand it at all, who just do some procedural kind of stuff and write their program as one huge class with a lot of methods in it; they try to do it in a C style. Then there are the people who go, "Ooo, objects! They're cool! Let's make gazillions of them!" Actually, that feels like almost a fault of the educational system, because if you look what lots of kids fresh out of college have been taught, it's, "Objects are good, define lots of them!" So you get programs with zillions of little adapters, with methods that are one or two lines long, and you start profiling them and they're spending all of their time doing method dispatching. Then the only kinds of optimizing compilers that do any good are the ones that spend almost all of their time trying to eliminate method dispatching and doing lots of full inlining.

Q: In your experience, how long does it take for a novice programmer to become a reasonably proficient [C/C++/Java] developer, capable of writing nontrivial production code? How long for a programmer with experience in one or more other languages? How can this time be shortened?

RITCHIE: I don't know the answer to this question either—my stock joke on similar ones is, "Well, I never had to learn C. . . . "

STROUSTRUP: That depends critically on the background of the novice, on the complexity of the task first attempted with C++, and on the teaching/learning approach.

For a novice programmer, a year and a half seems appropriate; for a programmer who is a novice to C++ and the techniques it supports half a year seems more likely. Clearly, I'm talking of the time needed to really use the facilities of the language in a significant application. Learning to write "Hello world" and its cousins can obviously be done in a few minutes.

I consider good libraries essential to provide a smooth learning curve and to properly sequence the learning of C++ concepts. For example, having the C++ standard library available makes it possible to learn the basic type, scope, and control structure concepts without having to deal with arrays, pointers, and memory management at the same time. Such fundamental, yet low-level concepts are best learned a bit later.

There is of course a danger in relying on libraries in teaching. They can provide an appearance of competence while hiding utter ignorance. One aim of teaching programming must be to make what is going on comprehensible rather than magical. To many programmers, the behavior of their system and even of their fundamental libraries

[6] B. Stroustrup. **"Learning Standard C++ as a New Language"** (C/C++ *Users Journal*, May 1999; also in *CVu* 12(1) January 2000).

is pure magic. This is dangerous. In this context, it is a strength C++ (and of C) that the standard libraries usually are implemented in the language itself using only the facilities available to every programmer.

GOSLING: I know that for somebody who is a pretty talented C++ programmer, an afternoon pretty much does it for lots of folks. You'll probably spend a lot of time going through the library manual. The language itself tends to be a snap to learn; it's all the library stuff that takes the time, and there the best way to do it is to just start writing and using it, and when you need something, look for it.

For people who have never written a program before, I don't know. There's been an interesting phenomenon: It used to be that the standard first language that people were taught in college was Pascal. A lot of that was because Pascal was relatively simple and clean, but also because when things go wrong in Pascal they tend to go wrong in somewhat more obvious ways. Just doing pointer checking and array bounds checking accounts for a huge fraction of the mistakes that novice programmers make, and so first-year courses tended to be pretty heavily dominated by Pascal. C didn't come until later just because it's so easy to do goofy things in C; it's so easy to have an array and write "`for( i = 1; i <= length; ... )`" and you have to explain, "no, it's not less than or equal to `length` and it's not `1`, it's `0` and it's less than." But of course your loop runs just fine, only maybe sometimes you'll discover that you smack the malloc header of the block immediately following the array and nobody's ever told you about that. The system didn't barf at you, yet the program will crash in different ways because it's very easy to corrupt somebody else's data structures, and so this one piece of the program looks like it's running just fine, this piece over there looks like it's broken, but the piece over there is running on data structures that have been corrupted by the first piece. That's just so incredibly common in C and C++ or any language that doesn't have a really strict memory model. In the last few years people have actually been doing a lot of teaching Java as a first course, because it has a lot of the safety that makes it easy to teach an initial course, plus unlike Pascal it's something that's actually commercially relevant as a career.

Looking Forward: Where Are We Going?

Q: Are there any important features missing from the C family of languages? From [C/C++/Java]?

STROUSTRUP: In which sense do C, C++, and Java constitute a family of languages? C and C++ has a large common subset, and over the years serious efforts have been made to minimize the inevitable tendency of the two languages to drift apart because they are controlled by separate standards bodies. Java, however, provides no real compatibility, and similar syntactic constructs have semantics different from the C and C++ versions. Clearly, Java borrows from C and C++, but under the skin, the similarities to Modula-3 seem greater.

Undoubtedly, all three languages could be significantly improved by addition. However, I see little reason to believe that there is a single major feature that would help all three languages.

GOSLING: There's certainly a proof by existence that there are no features that are essential that are missing, because people are able to get their jobs done. It's more about tuning the language than about what you can actually do. Some of the more obvious features are ones that would turn it into a different language. One of the things that always bugged me about C was the weak memory model, the fact that you could cast a pointer to a character into a pointer to an integer; it just did weird things. But then you change that and it's not C any more, it's basically Java.

If you look at the RFE logs for Java, there are really only two features that people ask for that show up with any frequency at all: One is assertions, and there's actually a group working on adding assertions to Java right now. The other is type polymorphism, something like templates, and Java actually has something like a poor man's template system right now in that the type hierarchy has a common root for anything and everything, namely Object, but there's also a group working on doing a proper job of type polymorphism in Java. It turns out to be a really hard problem. One of the reasons I left out type polymorphism in Java, even though I think it's a good idea, is that there's been a lot of argument in academic circles about the right way to do it. You'll find lots of individuals who have very strong opinions, and it's very hard to find anything like a consensus. Other issues are clearer: garbage collection is a good idea; goto is a bad idea.

I think that the languages themselves, for general applicability, are actually pretty complete. I think the interesting areas are as things get more specialized, and that has two forks: One is specialization in terms of building more libraries, and there are more libraries being built for Java today that anybody could list. The other is language features for particular communities. I actually think of operator overloading that's very much essential for people doing numerical work. People in the business end of things can actually make a reasonably strong argument for features that kind of look a little COBOL-ish, ones that have to do with database queries for example. If you construct database queries using the JDBC API, it's a little clumsy, the same way that constructing arithmetic expressions gets a little clumsy. In some sense to make it un-clumsy the system in a much deeper way has to understand the notation for databases and math.

Q: All languages change over time, and having a large installed base of users can both encourage and restrain such change. How has [C/C++/Java] managed change from release to release? What have you learned are the key considerations or techniques for successfully creating enhancements once you have a large installed base of programmers?

RITCHIE: I've learned that this is dreadfully hard, and that people involved with successful languages can become rabidly conservative no matter what their general inclinations. The problem is that once there is an installed base, even an "upward compatible" extension can be a real problem for many. Programs using the extension can't be handled by those not yet forced to upgrade. If they're lucky, a new bit of syntax will at least point out the problems because the old compiler won't accept it. If they are unlucky, they will have mysterious problems because the new language precisely defines something that was undefined before, and the new program depends on this definedness of the behavior.

STROUSTRUP: I think that the ideal strategy for evolution is to conduct extensive experiments before making a change, define new features precisely in a way that doesn't invalidate old code, and make them available everywhere. Then, after a year or two warn against older (banned or deprecated) features, and a year or two after that remove those features. Unfortunately, this strategy requires coordination between all vendors.

Older implementations have been a major drag on C++ programming style because they prevent people who want portable code from using recent facilities (in this context, "recent" sometimes means "invented less than 10 years ago"). It seems that the standard, and major users' wish to use the standard library, now provide significant incentives for implementers to conform and for users to upgrade to modern implementations.

Evolving a programming language and its implementations is inherently difficult because different people attach different importance of old code running without modification. I used to consider link compatibility far more important then source compatibility. Now, I'm not so sure. Many organizations do not have the resources to fix broken old code (even where it was always broken). This argues for the use of compiler options for controlling variations of compatibility.

I am no fan of compiler options for controlling variations of compatibility and warnings, but for some, they are necessary. However, the default setting on all C++ compilers should be full ISO C++ compliance, and departures from that ought to be somewhat bothersome. Unfortunately, people who care more about shipping deadlines than about code quality and genuine portability tend to argue for a language that is infinitely variable through options with the default being bug compatibility with some major vendor.

GOSLING: In general, we've been extraordinarily conservative about the language itself. The number of people who ask for changes to the language is rather small; the number of RFEs that have to do with language changes is a very, very small number. The place where change works and change is happening is in the libraries, because there's this established formalism for establishing what some people think of as the language, the whole class system where you can define new classes, and that does a pretty good job of encapsulating it, a really huge fraction of all the kinds of evolutions that people really would like.

Then there are incompatible changes, ones that alter the meaning of existing code. If the change changes the meaning of something in the library, pretty much uniformly the answer has been to change the name. The very first I/O libraries were reading and writing bytes, and that was mostly because we didn't have very much time to do it right. Really they should have been working on characters, not bytes; there's a huge difference between a character and a byte, and they have to do with internationalization and file encodings and ISO this or ANSI that or UTF-7 or UTF-8 and all this stuff that makes it much more complex. So rather than changing the semantics of input streams and output streams, we actually created new classes called readers and writers, and really left the old ones there for people who wanted to use them and the new ones for people who wanted to adapt to them.

With libraries, you can mostly get away with that. There are some places where it gets really hard to do that. In threads there have been all these issues about, "What does it mean to stop a thread? to kill a thread?" and there are many PhD theses to be written on the topic of, "What does thread.kill() mean?" because it cuts to the heart of a whole bunch of really thorny issues. And many systems, including Java straight out of the box, had this definition of what it means to kill a thread that were pretty expedient and kind of ignored a pile of issues. As the bug reports flowed in it became really clear that the whole notion of killing a thread was a bad idea.

Q: What things would you like to see become a part of [C/C++/Java] in the next 2–5 years? In the next 10 years? Why?

RITCHIE: For C, the new 1999 standard exists but has not become readily available, and it has quite a few changes from the earlier one, though not revolutionary ones. I think it needs to quiesce for a while.

STROUSTRUP: I think that the emphasis for C++ evolution should be in library building. I hope that the next standard will be primarily concerned with providing more facilities in the standard library and that language changes will arise primarily from the needs of such facilities.

Deciding what libraries to include, and keeping/making those coherent, will be very difficult. The ideal situation would be if someone came to the committee with a library that was general enough, elegant enough, efficient enough, and innovative enough to give the committee a base to work from. That was want happened with the STL. The committee isn't good at synthesizing things. I think it is the nature of committees to be better at polishing and completing than at actual design and experimentation. For that reason, I fear a stream of unrelated proposals for dealing with (small) individual problems.

Clearly, many people worry about concurrency and about interfaces to various systems (e.g. GUI, databases, operating systems, other languages, component models). This

is [the] most likely area of work and creating and maintaining consistency will be the major challenge. For example, it will be important to provide a consistent view of text from C++. Having some interfaces use C-style strings, others the standard library string, and yet others notions of strings derived from the systems being interfaced to is a recipe for chaos. I suggest that the standard library string be made the basis for all C++ bindings to "other systems." In general, we need a consistent view of resource management to permeate the standard library.

Q: Is there room and/or market demand for a new language in the C family?

RITCHIE: It's a little hard to imagine a language that's almost like C but enough better to displace it. This depends on what's meant by "C family." Does it mean '`uses` '? 'Names a type at the start of a declaration'?

STROUSTRUP: I think the user community would be best served by a single language providing low level support for systems programming. I think C++ would serve well there and I see no technical problems merging C and C++. I see many political problems, though. Vendors/purveyors like a multiplicity of languages and dialects so that they can claim advantages for their variant and lock-in their users. Also, every organization strives to perpetuate itself, so I don't see any of the languages, dialects, and variants dying peacefully.

Java doesn't provide support for low-level systems programming so it must rely on other languages (such as assembler, C, or C++). Conversely, for C or C++ to be useful for safe downloading into another machine, we need either hardware support (my preferred long-term solution for all languages), a C++ virtual machine (yes, such beasts exist), or a breakthrough in program verification (I'm not holding my breath).

So, is there room for another language beyond C, C++, and Java? There is certainly room for more languages. The question is whether such languages should be considered part of a family. Also, a new language should be produced only if there are significant problems that cannot be addressed by existing languages. Languages and dialects that provide only minor conveniences to users simply fragment the community and divert resources that could be used for the common good.

GOSLING: Oh, sure. I don't know what you call a "language in that family." Is any language that uses braces in the C family? Is AWK in the C family? You might call AWK in the C family. And I'd sure like to think that there's room for more languages; whether they're in the C family or not is almost irrelevant.

One of the real tragedies of the years before Java came out was that programming language research had almost stopped worldwide. Then Java came out and Java was actually successful. When I've gone to universities, one of the comments I've gotten a lot was, "Wow, you've really legitimized the whole study of programming languages." I think it would be a tragic statement of the universe if Java was the last language that swept through. I would hope that some new more interesting paradigms would come out, and whether I'll be involved in that or not, who knows. It would be nice if the world didn't wait another 20 years. Maybe another five years would be nice to get some stability, because changing the programming language is like changing the underlying genetic structure of an organism. It's about as deep in the core of the craft of programming as you get, so it really is difficult to change it.

Q: Which types of applications are well suited for [C/C++/Java]? Which are not?

STROUSTRUP: Without supporting libraries, most serous applications are unnecessarily hard in C++. With suitable libraries, most are reasonably easy. C++ has inherent strength in applications with a systems component, especially where there are constraints on resources (such as run time and memory). From that base, C++ provides significant support for organizing larger programs for easier maintenance and evolution.

When suitably supported by libraries, C++ can be an excellent teaching language. In a teaching concept it is essential to start out with modern C++ rather than getting students wrapped up in old styles. See "Learning Standard C++ as a New Language" (that paper can be downloaded from my homepages). In teaching, C++'s greatest strength is probably that it allows the student to become acquainted with a variety of techniques in a from that can be applied to real-world problems.

GOSLING: Java is certainly well-suited for anything that involves networks and in places where security is a concern. There are some things that it really was never targeted for, although people are certainly doing them. Writing device drivers tends to be kind of awkward, although it's surprising how many people have gone and done a number of fairly obvious things to make device driver writing in Java more straightforward. Also, there are people doing really intensive numerics . . . Java wasn't really designed with that as a focus, and there are some things about it that are somewhat complicated, yet there are many people who do numeric programming in Java and find that the tradeoff is worth it.

Q: Are languages getting easier or more difficult to learn and use? Do you see any progressive usability trends among C, C++, and Java as those languages have appeared and/or added features over time?

STROUSTRUP: Languages are getting easier to use. However, it may not seem to be so because we are trying to achieve far more difficult objectives these days. For example, C++ with standard-library facilities (such as string, vector, and algorithms) is far easier to use than C++ using C-style strings, arrays, and C standard library functions to solve the same problems.

I see a definite trend in ease of use and expressive power from C to C++. After all, with minor exceptions, C++ is a superset of C. However, I don't see such a line from C++ to Java. Java attempts to restrict programmers to a single style of programming. This makes Java easier to use within that domain, but leads to ugly workarounds when the edges of that style are approached. The need for casting when using Java containers is an obvious example.

To me, it seems as if C++ and Java are on divergent courses as far as style and language evolution are concerned. One might expect that heavy use of libraries would minimize that divergence, but with modern C++ libraries emphasizing sophisticated uses of templates and modern Java libraries emphasizing inner classes, that does not appear to be the case.

Often discussions of "ease of use," "learning curve," and "complexity" are confused because often a concept can be represented in a language, in a standard library, in a library that is not part of the standard, or in code written by the application programmer. As a rule of thumb, I prefer to have difficult issues addressed in the language or in the standard library. By placing "complexity" there, many programmers can benefit from solid work of experts and we minimize the need to "reinvent the wheel." However, a language is perceived as complex if it provides such support, and simple if it leaves the problems to the individual application builder.

GOSLING: I certainly see a trend in the form of people being concerned about usability. To what extent they're actually achieving usability is another question altogether. Actually, I don't think that usability is a really great word, because it has this connotation of something that's trivial, in the sense that it's something that isn't necessary, it's something that you do just to make yourself more comfortable, like adding a pillow to a chair.

For me, the big thing that's going on is that the complexity of the systems that we all build is getting larger and larger every day. Certainly all the hardware that we build this stuff on is being driven by some flavor or derivative of Moore's Law. The software systems we build aren't following a curve anything like that, but they're definitely getting

more complex—and quite quickly. How do you build big, complex systems? There are lots of things that we don't understand really well, and I think that actually a lot of these are beyond the realm of what people would largely think of as the programming language.

One of my favorite examples these days is Bresenham's line drawing algorithm. You pick up any first-year graphics book and they'll describe Bresenham's algorithm for drawing a line, and it's about five statements long, does what it does, and it's pretty simple. You go and you take a look at any industrial-strength implementation of Bresenham's algorithm and it's thousands if not tens of thousands of lines long. Why is it so big? Why is there all this complexity in what is really a very simple algorithm? A lot has to do with the different special cases, understanding more what's going on beneath the machine, things like the cache line size; does this machine have cache lines that are 16 bytes wide? 8 bytes wide? 32 bytes wide? You end up wanting to optimize the inner loop differently. What is the bit depth of the pixel? Are you talking about 1-bit pixels? 8-bit pixels? 16-bit pixels? 15-bit pixels? Are you painting the pixel? XORing the pixel? alpha-blending the pixel? A lot of these things actually can be expressed in terms of algorithm transformations, and I did a project about 10 years ago building a system that was kind of a macro preprocessor for C, but it was more like a theorem prover. It was sort of a macro preprocessor on the semantic graph, and you could write macros on the semantic graph. You could write algorithm transformations, and so you really could write Bresenham's algorithm in five lines and express transformations and get it up to the industrial-strength version. One of the big advantages of doing it that way is that you've got a better handle on whether or not it's correct, and the whole testing and tuning problem gets a lot easier. And yet, is that a programming language thing or is it a programming environment thing? In some sense you haven't changed the semantics of the language much at all in terms of what you can do. You've changed how you say it, and you're doing manipulations not so much of the text of the program but of the semantic graph. There's just huge amounts of territory in that sort of thing. That system that I built was used for a lot of Sun's graphics algorithms, but I had this huge problem that I couldn't get people to understand what it did. It was seriously weird, and you could write some very cool programs with it, but getting people to understand what it meant to do theorem proving as part of the construction of a new algorithm was pretty tough. A lot of designing programming languages is not so much about, "What's a cool feature? What fits some fancy academic criteria?" It's really about, "What fits with the developers?"

Personal Preferences

Q: We all have different reasons why we've chosen to be in the software industry. What was it that drew you to the software field? What makes it cool?

RITCHIE: I started out interested in physics, and still maintain an amateur interest in keeping up with what's happening at its edges. Sometime in college and early grad school, I spent a lot of time in theoretical computer science (Turing machines, complexity theory). Meanwhile I also became more fascinated with real computers and, I suppose, the immediacy of the experience they provided: when you write a program, you can see what it does right away. All of these things connect with each other in interesting ways. Being involved with this sort of activity was what motivated me. Somehow I didn't think of what I was doing as joining the Software Industry, although, even in 1968, I guess it was.

STROUSTRUP: I'm not sure exactly what brought me to computers. I saw computers as a practical and useful outlet for scientific interests. I really like building things. When you build you get feedback from the tools, from the program/system, and from users. That's

what pushes the imagination beyond what is obvious and beyond the preconceptions and doctrine of an individual, an academic field, or an organization. I like Kristen Nygaard's notion of programming as a means of understanding something.

GOSLING: I don't know what it is about my genetic structure, but I just like to build stuff. In some sense whether I'm building software or building a chair or building dinner—also known as cooking—I get a kick out of just creating stuff. One of the things that's nice about software is that you can create just about the most amazingly sophisticated complex things and you can do it fairly quickly. If I was a watchmaker, I'm sure I'd get frustrated with how hard it was to create things with lots of gears and pulleys; and yet, just looking at a good watch, you take off the back and it's just so cool. I don't know why it's cool, it just feels really cool to me. With software you can put as many as gears and cams and whatever in there as you want, and things just get complex. In some sense the thing I like most about software is the thing I end up spending most of my time fighting against, which is complexity.

Q: What was the first programming language you ever used?

RITCHIE: I can't quite sort out the times. More or less simultaneously, around 1962, I went to a non-course talk about Cobol (by Jean Sammet), and took a course that involved programming both analog computers using a plugboard and writing Univac I machine language (no assembler). Also about this time, I visited the IBM office in Cambridge (MA) and they gave me manuals about Fortran, which I read avidly.

STROUSTRUP: My first programming language was Algol60 on a GIER computer. The GIER was a 1960s Danish computer with 1K 42 bit (I think, plus a few extra bits for hardware testing) words.

GOSLING: The first programming language I ever used was a language called Focal 5. Focal stood for "formula calculi." It was a scripting language for PDP-8's. I wrote most of my earliest programs using Focal 5. It's a language whose complete compiler and runtime system can be implemented in under 24 hours. It's actually a nice little language, way simpler than Basic.

Q: What languages, or features of languages, inspired you?

STROUSTRUP: In addition to Simula67, my favorite language at the time was Algol68. I think that "Algol68 with Classes" would have been a better language than "C with Classes." However, it would have been stillborn.

GOSLING: They're all over the map. Using Lisp, the thing that influenced me the most was the incredible difference garbage collection made. Using Simula and being a local maintainer of the Simula compiler was really what introduced me to objects and got me thinking about objects. Using languages like Pascal got me really thinking about modeling. Languages like Modula-3 really pushed things like exception mechanisms. I've used a lot of languages, and a lot of them have been influential. You can go through everything in Java and say, "this came from there, and this came from there."

Q: What was the first program you ever wrote? On what hardware?

RITCHIE: As in the above, the first on a real computer was the Univac I program. (As I learned a year or so later, the exercise was in fact to implement some of the APL operators—Iverson was around then).

STROUSTRUP: I don't recall, but it must have been some computer science 101 exercise in Algol60 on a GIER computer. The first programs that I did for real—that is, for others to use—were business programs written in assembler for Burroughs office computers. I financed most of my Danish masters degree that way.

Q: Is there a programming language that is the best choice for all (or nearly all) application development? If yes, which language is it, and what makes it best? If not, what would it take to create such a language?

RITCHIE: No, this is silly.

STROUSTRUP: No. People differ too much for that and their applications differ too much. The notion of a perfect and almost perfect language is the dream of immature programmers and marketeers. Naturally, every language designer tries both to strengthen his language to better serve its core community and to broaden its appeal, but being everything to everybody is not a reasonable ideal. There are genuine design choices and tradeoffs that must be made.

GOSLING: I think the one that has the best broad coverage is Java, but I'm a really biased sample. If you're doing things that are heavily into string pattern-matching, Perl can be pretty nice. I guess actually those are the ones I use much at all these days. Most of the older languages are completely subsumed; the reasons for using some of them are more historical than anything else.

Q: Programmers often talk about the advantages and disadvantages of programming in a "simple language." What does that phrase mean to you, and is [C/C++/Java] a simple language in your view?

RITCHIE: C (and the others for that matter) are simple in some ways, though they are also subtle; other, somewhat similar languages like Pascal are arguably simpler. What has become clear is that aspects of the environment like libraries that aren't part of the core language are much bigger and more complicated. The 1999 C standard grew hugely more in the library part than in the language; the C++ STL and other things are big; AWT and other things associated with Java are too.

STROUSTRUP: I see three obvious notions of "simple:" to be easy to learn, to make it easy to express ideas, and to have a one-to-one correspondence to some form of math. In those terms, none of the three languages is simple. However, once mastered, C and C++ make it easy to express quite complex and advanced ideas—especially when those ideas have to be expressed under real-world resource constraints.

GOSLING: For me as a language designer, which I don't really count myself as these days, what "simple" really ended up meaning was could I expect J. Random Developer to hold the spec in his head. That definition says that, for instance, Java isn't—and in fact a lot of these languages end up with a lot of corner cases, things that nobody really understands. Quiz any C developer about unsigned, and pretty soon you discover that almost no C developers actually understand what goes on with unsigned, what unsigned arithmetic is. Things like that made C complex. The language part of Java is, I think, pretty simple. The libraries you have to look up.

Q: What topics do you feel are missing from university computer science and engineering programs around the world that would help to improve the quality of software?

STROUSTRUP: In some well-respected computer science departments, you can graduate without having written any code. That ought not be possible. Nobody should graduate with a degree in computer science or computer engineering without having completed a significant programming project. Code is the base of computing and people without a "feel" for code tend to seriously misjudge what skills, tools, and time are needed to build good systems.

Much shoddy design and programming have root causes in misconceptions about what constitutes good code. I think reading and writing code should be a significant part of the training of every computer professional—even in the education of decision

makers who do not program for a living. However much we talk about "Computer Science" and "Software Engineering", building systems still has a large practical component relating to reading, writing, and maintaining code, and this will not change in the foreseeable future.

No, I'm not arguing that we should just teach hacking. I argue for a balance between theoretical and practical skills. Other parts of the educational establishment have the opposite problem. They train people in practical skills (only) rather than educating them.

GOSLING: The number one thing that's missing has been a dose of reality. People have been getting a pretty good education in the technology; what they tend to not get is a good view of what goes on in an actual engineering organization: How do you deal with the interpersonal dynamics of people? If you've got some large project and you parcel up the pieces to different people, how do the people relate to each other? How do you deal with bug fixing, when most of what most developers do is not writing new code, but almost always what you're doing is fixing bugs in something that's already there? How do you deal with a manager who's a jerk? A lot of education courses are really around just the task of writing software.

Q: Outside the C family of languages, what language(s) do you like the best? What in particular makes them interesting to you?

RITCHIE: I have to admire languages like Lisp. But, just as in the earlier question about a "simple" language, an astonishingly simple language grows in practice into something that can be fearsome.

STROUSTRUP: Algol68, CLOS, and ML springs to mind. In each case, what attracts me is flexibility and some very elegant code examples.

Q: What are your favorite software-related books?

RITCHIE: This is self-serving at second hand, but the works of Kernighan and Pike (separately or collectively) are nice.[7,8] In some ways I like the old Kernighan and Plauger on Programming Style[9] best of all, if for no other reason than the huge joke involved in taking elsewhere-published programs and showing how bad they were. The software book I enjoyed most is Ben Ross Schneider's *Travels in Computerland*,[10] also old and now hard to get.

STROUSTRUP: Brooks' *The Mythical Man Month*,[11] Booch's *Object-Oriented Design*,[12] and Gamma et.al.'s *Design Patterns*.[13]

GOSLING: One of my favorites from early on was Bill Wulf et al.'s book on the design of an optimizing compiler. I think that one's been out of print for 20 years, but it's still a really great book, and if you ever want to know actually how to write a compiler it's the best book that's ever been written as far as I can tell. Then there's Jon Bentley's

[7] B. Kernighan and R. Pike. **The Practice of Programming** (Addison-Wesley, 1999) ISBN 020161586X.

[8] B. Kernighan and R. Pike. **The UNIX Programming Environment** (Prentice Hall, 1984) ISBN 013937681X.

[9] B. Kernighan and P.J. Plauger. **The Elements of Programming Style** (McGraw-Hill, 1988) ISBN 0070342075.

[10] B. Schneider. **Travels in Computerland: Or, Incompatibilities and Interfaces: A Full and True Account of the Implementation of the London Stage Information Bank**, ASIN 0201067374.

[11] F. Brooks. **The Mythical Man-Month: Essays on Software Engineering** (Addison-Wesley, 1995) ISBN 0201835959.

[12] G. Booch. **Object-Oriented Analysis and Design With Applications** (Addison-Wesley, 1994) ISBN 0805353402.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. **Design Patterns : Elements of Reusable Object-Oriented Software** (Addison-Wesley, 1995) ISBN 0201633612.

Programming Pearls,[14] which is a wonderful book and I just wish that he would write a lot more. I really like Sedgewick's *Algorithms* book.[15]

Q: If you could recommend two books on [C/C++/Java] besides your own, what would they be and why?

STROUSTRUP: *Design Patterns* again, for people aiming at flexibility. To complement, I'd like to recommend a book focussing on efficiency, but I don't know one I'm completely comfortable with. For many people Koenig and Moo's *Ruminations on* C++[16] would do a valuable service in shaking their preconceptions on what C++ is and can be.

GOSLING: I'm one of the worst people to ask about Java books, because I basically don't read any Java books. Mostly what I read is the API documentation that's on the web.

Programming Language Standards

Q: Is it important to have a formal (e.g., ISO/ANSI) standard for a programming language? What are the advantages? The disadvantages?

RITCHIE: At some point a formal standard tends to become necessary, particularly for language that evolved informally (certainly C, but also C++ and Java). The process does tend to pin things down somewhat and also brings to the table some changes or expansions that others besides the designers might have envisioned. A written standard also gives organizations (in all sectors) a certain amount of assurance that their technical folk, who might have a fascination with a language or other bit of software, aren't just following some will-of-the-wisp. There is a certain gravitas that attaches to the process.

At the same time, there are disadvantages, in that whatever clear (or cloudy) vision that the original designers might have had ends up colored by whatever strange proposals the participants in the process bring forth. Complication invariably results as a result of the compromises.

STROUSTRUP: In today's overheated commercial atmosphere it is essential to have a formal standard. Without a formal standard, there is no defense against rapacious vendors. Naturally, an ISO standard is not a complete defense, and not all of our languages, libraries, or tools can be standardized. However, without a standard, users are completely at the mercy/whim of their suppliers.

The major advantage is to have a standard that isn't manipulated to serve the commercial interests of a company or a small group of companies. An ISO standard emerges from a consensus process that gives voice to individuals, companies, and nations.

The major disadvantages are that the essential consensus building takes time, that the committee has limited resources, and that it is hard to create a consensus for something innovative. However, it can be done: Remember the STL.

GOSLING: I think it would be a nice thing to have a formal standard. I think most people have a very naive view of what a formal standard is. Fundamentally the standards bodies that create formal standards are places to store documents that have stamps on them. They don't have any notion of conformance testing. To be an ANSI standard C compiler, all you have to do is say that you're an ANSI standard C compiler; there's absolutely no testing of that at all. The hard part for me about formal standards is that, to an extent that most people are completely unaware of, the formal standards process is intensely

[14] J. Bentley. **Programming Pearls, Second Edition** (Addison-Wesley, 1999) ISBN 0201657880.

[15] R. Sedgewick. **Algorithms in C, 3rd edition** (Addison-Wesley, 2000) ISBN 0201849372. (There are at least ten "Algorithms" books by Sedgewick, covering different scopes or specialized for different programming languages. This is one of the most recently written/updated.)

[16] A. Koenig and B. Moo. **Ruminations on C++** (Addison-Wesley, 1996) ISBN 0201423391.

political. We've made a couple of runs at turning Java into a formal standard, and the politics got so absurd that it became impossible in both cases.

STROUSTRUP: Conformance suites are widely used for ISO C and ISO C++.

Q: Is it important to have a de facto standard for a programming language? What are the advantages? The disadvantages?

STROUSTRUP: A de facto standard is a major advantage to its owner and its owner's friends/allies. It can also benefit third parties, such as professors, students, and small companies. As long as the corporation owning a de facto "standard" is in the process of eliminating alternatives, tools and support tend to be cheap and marketing clout affects the intellectual and business atmosphere. Later, the price goes up.

GOSLING: I think that in a strong sense de facto standards are the ones that really matter. It hardly matters what's actually written down in anybody's standards document; what actually matters is what people have actually implemented. Where de facto standards get sticky for me is that when there are multiple implementations of de facto standards they tend to be very variable. You get things like Unix where a lot of things call themselves Unix and things just don't move back and forth. I'm a real fan of Linux; the problem is that there are so many flavors of Linux and they're all just different enough that life is really hard.

Q: What have you learned from going through a standards effort for [C/C++/Java]?

STROUSTRUP: Consensus building is tedious and necessary. People and organizations really do want to do "the right thing" given half a chance. Political problems cannot be solved. The way you handle a political problem is to find the technical problem at its root and solve that instead. Then, the political problem will evaporate.

GOSLING: It's mostly been a political education that I would rather not have ever had, thank you.

The "Century 21" Software Industry

Q: Prognostication is hard, but valuable even if it can never be 100% accurate. In your view, what are the major forces driving the kinds of software we will be writing in the future? and the way we will be writing that software?

RITCHIE: The most obvious change that has occurred recently is the increase in use of "scripting languages" that are generally interpreted. These range from the basic ones like HTML to Perl, Tcl/Tk, and Javascript, and then to presentation-graphics packages for making WWW pages or slides.

The number of people doing computing in the traditional sense is increasing somewhat, but the numbers who are making things for others to look at is expanding enormously.

STROUSTRUP: The future? these days it is close to impossible to even describe the present.

We have to see more emphasis on correctness, quality, and security. Our civilization depends critically on software, and we have a dangerously low degree of professionalism in the computer fields.

GOSLING: I think that the absolute two driving forces are complexity, and the environment in which that software is running because it is changing so much. It is no longer the case that people are writing software that just runs on the mainframe. Things have been spreading out for years into all kinds of esoteric places, and seeing what the programming environment looks like inside a smart card or a cell phone—these things are

more and more important. The systems that we build have very much end-to-end character where a developer can't just focus on the one machine in the network, they have to focus on the whole system; and that system is environments spread out all over many places, so it has a lot of complexity, and this complexity and diversity of environments is just exploding.

In terms of the kinds of software we write, it's rather interesting when you think about the problem you're trying to solve. You look at a piece of software, and see how small a portion of the software is actually devoted to solving the problem. It's getting to the point where it's almost infinitesimally small, because the rest of the software is all about: How do you do the user interface? How do you talk to the network? How do you deal with error recovery? How do you deal with reliability? How do you deal with all these peripheral issues that are all around the core that is your actual problem? An interesting example application is Quicken, the checkbook balancer. This thing that maintains your checkbook is really simple. And yet you look at all the goo around it, what it takes to make all that reliable, easy to use, integrate with the environment, and so on, it's all this peripheral stuff that's really soaking up the cycles.

STROUSTRUP: Too often, we simply throw more people at a problem. I think that in the longer run, we simply have to take a more systematic approach to our system development, deployment, and maintenance. An increase in professionalism will be necessary and will allow us to use more advanced tools and techniques. There is a tendency to dumb down languages and tools so that "programmers" with a few weeks of "education" and "experience" can start using them. I think this trend has to be reversed. We need tools for professionals.

In languages, I think this will imply a move towards languages that are better specified, more flexible, and better suited to analysis. Maybe the synthesis of object-oriented and functional techniques will finally happen.

I don't think that languages will be either interpreted or compiled. The compile vs. interpret decision will be made for individual programs and parts of programs as dictated by practical needs.

I think that we'll have something that is recognizably programming as we know it today. There will be professional programmers who struggle with code in a variety of representations and use tools to read, analyze, and write such code. However, I suspect the vast majority of software will be created by people who are experts in other fields or are simply reorganizing their personal environment. What such non-professionals will use, I have no idea of, but it will rely heavily on a very supportive infrastructure provided by the professional systems builders—and it will not be perceived by its users as anything like code. I suspect it will be highly declarative and rule based.

I really hope 2050s software and systems will be too advanced for me to imagine today, just as a 1950s expert had little chance of predicting today's systems with any accuracy or in any detail.

GOSLING: One of the things I've found kind of depressing about the craft of computer programming is that you see all these fancy development environments out there, and they tend to be targeted at solving certain specific kinds of problems, like building user interfaces. But if you go around and look at the high-end developers, the people who actually have to implement algorithms, if you're using one of these high-end IDEs the IDEs generally don't help you much at all, because they drop you into this simple text editor. The number one software development environment for high-end developers these days is still essentially EMACS. At their heart, these tools are 20 years old, and there hasn't been much in the way of dramatic change. People have made all kinds of stabs at graphical programming environments and that, and they've tended to be failures for one reason or another.

It's always felt to me like there have been some good ideas in these exotic development environment ideas. Why is it that they tended to not work? Why is it that people still use ASCII text for programs? It just feels like there's so much territory out there that's beyond the bounds of ASCII text that's just line after line, roughly 80 characters wide, mostly 7-bit ASCII, something that you could type in on a teletype. That's still where programming basically is these days, and it's proven to be a very hard thing to get anything beyond that.