From C to C++: Interviews With Dennis Ritchie and Bjarne Stroustrup

By Al Stevens, January 01, 1989

In these exclusive interviews, Al Stevens talks with language pioneers Dennis Ritchie and Bjarne Stroustrup about where C and C++ came from and more importantly, where they might be going.

The interview with Dennis.

Bjarne Stroustrup is the creator of C++, the object-oriented extension to the C language. He is a researcher at the AT&T Bell Laboratories Computing Science Research Center where, in 1980, he began the development of the C++ extensions that add data abstraction, class hierarchies, and function and operator overloading to C. The C++ language has undergone several versions, and the latest is Version 2.0. Dr. Stroustrup maintains an active presence in all matters concerning the development, advancement, standardization, and use of C++.

DDJ: Many experts are predicting that C++ will be the next dominant software development platform, that it will essentially replace C.

BS: They're not alone. People were saying that five years ago.

DDJ: When you conceived the idea of C++ as an extension to the C language, were you thinking about object-oriented programming in the way it's come to be known, or were you looking to build a solution to a specific programming problem that would be supported by the features that you built into C++?

BS: Both. I had a specific problem. All good systems come when there is a genuine application in mind. I had written some simulations of distributed computer systems and was thinking about doing more of them. At the same time I was thinking about the problem of splitting Unix up to run on many CPUs. In both cases I decided that the problem was building greater modularity to get fire walls in place, and I couldn't do that with C. I had experience with Simula, writing rather complex simulations, so I knew the basic techniques of object-oriented programming and how it applied.

To solve the problem I added classes to C that were very much like Simula classes. That, of course, was the solution to a particular problem, but it included a fair amount of general knowledge about techniques, thoughts about complexity and management, complexity of modularity and all the baggage that you get from Simula. Simula is not ad hoc, especially not when it comes to the class concept, which is what I was interested in.

DDJ: Are you familiar with any of the PC ports to C++, specifically Zortech C++, Guidelines C++, and Intek C++?

BS: Only from talking to people and listening to discussions about them. They all sound good. The CFRONT ports, Intek and Guidelines, have the advantage of having the same bugs and features that you have on the bigger machines all the way up to the Cray, whereas Zortech has the advantage of being native to the PC world.

I walked around back in 1985 explaining why the current implementation of C++ couldn't be put on a PC. I designed the language under the assumption that you had one MIPS and one Meg available. Then one day I got fed up with explaining why it couldn't be done and did it instead. It was a pilot implementation, and it wasn't ever used, but I proved that it was possible, and people went and did the real ports. All the implementations are reasonably good, and they could all be better. Given time, they will be.

DDJ: Do the PC ports accurately implement C++ the way you have it designed?

BS: We do have a problem with portability from one machine to another. If you have a large program of ten to twenty thousand lines, it's going to take you a day to move from one independent implementation to another. We're working on that. Standardization is beginning. We're all sharing language manual drafts, and so it's trying to pull together. But a large program port will still take a day as compared to the ANSI standard ideal where you take something from a PC to a Cray and everything works. Of course, you never really get to that point even after full standardization.

DDJ: There are lots of rumors about Borland and Microsoft coming out with C++ compilers. Has any of this come to your attention?

BS: I've talked to people both from Microsoft and from Borland. They're both building a C++ compiler, and it sounds as if they're building it as close to the 2.0 specification as they jolly well know how to. Naturally, for their machines they'll need something like near and far, which is not standard language, but that's pretty harmless.

Both asked for a bit of advice and Microsoft asked for the reference manuals. I've talked to the Borland guys. I'm sad to say they didn't ask for a manual, but maybe they got one from other sources. The PC world is pretty cut-throat. Maybe people get the impression everybody is cut throat. That's not quite the case.

DDJ: One of the advantages of languages such as C and C++ is that they can be implemented on a wide range of machines ranging from PCs to Crays. With more and more people using PCs in their work, it's widely believed that acceptance in the PC world is what spelled the overwhelming success of C as the language of choice.

BS: That's widely believed in the PC world. In the minicomputer world it's widely believed that the PDP-11 and the VAX spelled the success of C and that is why the PC world picked it up. One of the reasons C was successful is that it was able to succeed in very diverse environments. There are enough languages that work in the PC world, enough that work in the minicomputer world, enough that work on the large machines. There where not very many languages that worked on all of them, and C as one of them. That's a major factor. People like to have their programs run everywhere without too many changes, and that's a feature of C. And it's a feature of C++.

DDJ: Do you see the PC as figuring as prominently in the acceptance of C++?

BS: Definitely. There are probably as many C++ users on PCs as on bigger systems. Most likely the number of PC users will be growing the fastest because the machines are smaller. People who would never use a PC for their professional work -- there are still a lot of those -- nevertheless like to play with things on a PC to see what it is, and that is where PCs come in. Similarly, if you are working on a PC, sooner or later you run into a bigger machine, and it's nice to be able to carry over your work. I'm very keen on portability.

DDJ: Do you have opinions as to whether preprocessing translators, such as the CFRONT implementation on Unix, have advantages over native compilers such as Zortech C++?

BS: It depends on what you're trying to do. When I built C++ I felt that I couldn't afford to have something that was hard to port, meaning it mustn't take more than a couple of days. I thought if I built a portable code generator myself, it would be less than optimal everywhere. So, I thought if I generate C, I could hijack everybody else's code generators.

For the last 40 years we've been looking for a universal intermediate language for compilation, and I think we've got it now, and it's called C. So, what I built was something that was a full compiler, full semantic check, full syntax check, and then used C as an intermediate representation. In essence I built a traditional two-pass compiler. And two-pass compilers have great advantages if you've got lots of code generators sitting around for them, and I had. They have the advantage that they tend to find errors faster than one-pass compilers because they don't start generating code until they have decided that the program looks all right. But they tend to be slow when they actually generate code. They also tend to be slightly larger and slightly slower throughout the whole process because they have to go through the standard intermediate form and out again.

And so, the advantages for the translator technology are roughly where you have lots of different machines and little manpower to do the ports. I see the one-pass compilers, the so-called native compilers, useful for machines and architectures where the manpower available for support and development is sufficient to make it worthwhile, which is when you've got enough users.

The two-pass strategy for translators was essential in the early days and will remain essential as long as new machine architectures and new systems come on the market so that you need to get a compiler up and running quickly. As the C++ use on a given system matures, you'll see the translators replaced by more specifically crafted compilers. On the PC, for example, CFRONT likes memory too much; it was built for a system where memory was cheap relative to CPU time. So once you know you are working for a specific architecture, you can do intelligent optimizations that the highly portable strategy that I was using simply mustn't attempt.

DDJ: Are there different debugging considerations when you are using a preprocessing translator?

BS: One of the things that people have said about the translators is that you can't do symbolic debugging. That's just plain wrong because the information is passed through to the second pass and you can do debugging of C++ at the source level. Using the 2.0 translator we're doing that. That 1.2 versions didn't have quite enough finesse to do it, and people didn't invest enough in modifying debuggers and the system-build operations to give good symbolic debugging. But now you have it.

DDJ: Can you estimate the worldwide C++ user base today?

BS: Fifty thousand plus, and growing fast, and that is a very conservative estimate.

DDJ: Have you formed plans to rewrite any or all of Unix with C++?

BS: Unfortunately, I haven't, despite that being one of my original thoughts. I've been bitten trying to write software that was too complex for the tools I had. When thinking about rewriting Unix, I decided that C wasn't up to the job. I diverted into tool building and never got out of that diversion. I know that there is an operating system written in C++ at the University of Illinois. It has a completely different kernel, but it runs Unix and you can make it look like System V, USD,

or a mixture of the two by using different paths through the inheritance trees in C++. That's a totally object-oriented system built with C++. I know that AT&T and Sun have been talking about Unix System V, Release 5, and that there are projects working on things like operating systems rewrites, but whether they become real or not depends more on politics and higher corporate management than anything else. Why should we guess? All we can do is wait and see.

DDJ: Is what you do now primarily related to the development of C++ or the use of it?

BS: Both. I write a fair bit of code, still. I do a lot of writing, and I coordinate people, saying "Hey, you need to talk to that guy over there," then getting out of the loop fast. I do a fair bit of thinking about what else needs to be done with C++ and C++ tools, libraries and such.

DDJ: C is a language of functions, and a large part of the ANSI C standard is the standardization of the function library. C++ has all that as well and adds classes to the language. Is there a growing library of C++ classes that could eventually become part of a standard?

BS: The problem is there are several of them. We use some inside AT&T, and several of the other purveyors of C++ compilers and tools have their own libraries. The question is to what extent we can pull together for a standard library. I think that we can eventually get to a much larger standard library and much better than what is available and possible in C. Similarly, you can build tools that are better than what is possible with C because there is more information in the programs.

But people, when they say standards, tend to think about intergalactic standards, about things that are available in any implementation anywhere, and I think that they think too small. There are good reasons for differences between the ideal C++ environment for a Cray and the ideal C++ environment for a PC. The orientation will be different as will the emphasis on what is available. So we will see many standards, some for machine architectures, some over ranges of machines, some national standards. You could imagine the French having a whole series of libraries and tools that would be standard for people doing French word processing, for instance. You will see national standards, international standards, industry standards, departmental standards. A group building things like telephone operator control panels would have the standard libraries for everybody in the corporate department doing that kind of work. But a token standardization of everything, you won't see. The world is simply too big for that. But we can do much better than we're doing now.

DDJ: To the programmer, there is an event-driven or object-oriented appearance to the graphical user interfaces (X Windows, the Macintosh, Presentation Manager, MS-DOS Windows, and so on). These seem to be a natural fit for the class hierarchies of C++. How would these facilities be best implemented, and do you know of any recent efforts in these areas.?

BS: Some people have the idea that object-oriented really means graphics because there is such a nice fit. That has not been my traditional emphasis. The examples people have seen of object-oriented programming and object-oriented languages have, by and large, been fairly slow. Therefore, their use has been restricted to areas where there's a person sitting and interacting. People are relatively slow.

My first aims were in areas where you had complex programs that had a very high demand on CPU and memory, so that's where I aimed first. But people have been building very nice toolsets for doing user interfaces with C++. There is one from Stanford called "Interviews." Glockenspiel, in cooperation with Microsoft, is selling Common Views, which is a C++ toolset that looks and feels exactly the same whether you are under Presentation Manager, MS-DOS Windows, or on a Mac. There are C++ libraries for Open look.

The problem with all these so-called standards is that everybody seems to have their own standards, and then you start wondering how you can get toolsets that give platform independence across all of these I think that's one place where C++ comes in. Most of the differences between the major systems in the areas of text handling -- as opposed to high-performance graphics -- seem to be quite manageable as a common set of classes that could be standardized at the language level. It's certainly something that's worth exploring because the world is getting more fragmented.

DDJ: Are you familiar with Objective C, a C language extension that appears to do at least a subset of what C++ does, and how do the two languages compare?

BS: Vaguely. The company that sells it will claim that it does a superset of what C++ does, and that whatever C++ does that it does not, is not as important, naturally, I disagree. There is much higher emphasis in C++ on static type checking and on coherence of the type system. C++ is a rather large affair with multiple purveyors and multiple libraries, where objective C is a corporate language from a corporation that wants to make its fortune out of it. That places a different emphasis on everything.

DDJ: Concurrent C and Concurrent C++ are, like C++, extensions to the language. They add parallel processing operations to C and C++ for the development of multitasking programs that are portable among multitasking platforms. Do you have any comments on Concurrent C++? Is there a need for portable parallel processing, and do you think that Concurrent C++ fills that need?

BS: I don't like the idea of putting ADA tasking into C or C++. I think it solves the issues dealing with concurrency at the wrong level, sort of a medium-level thing. It doesn't give the transaction processing view and transaction logging that you need in data bases. It doesn't give the machine near world that you need when you write an operating system kernel or a real-time application. Personally, I don't like that approach at all. The approach I've taken with C++ is to provide con currency in the form of libraries. We have the Task Library that provides a much lower-level system that allows you to write multi-threaded programs. I've used it for simulations where I needed a couple of thousand processes or tasks and I want them to run with fairly minimal overhead. It has been used for robotics and such. I much prefer the library route over the route of adding syntax to the language. I think it serves more people better.

DDJ: Let's discuss the future of C++ and what programmers can expect in the next decade. The immediate future of C++ is, of course, 2.0, which adds multiple inheritance to the language. Can you summarize the other features added by version 2.0?

BS: It's a reworking of the language, polishing off the little rough corners, the unnecessary restrictions, and the problem areas we found. Even multiple inheritance can be seen as removing a little odd restriction, which was that you couldn't have more than one base class. We can argue how major an extension it is. Some people think it's major. I think it's sort of medium. It allows you to do things that you could do with C++ but noticeably cleaner and easier. I don't think it allows you to do anything radically new, and most of the other features I've added to 2.0 are of that ilk. It's meant to stabilize the language, it's meant to increase the quality of the implementations, and it's meant to remove unnecessary restrictions without destroying run-time or space efficiencies.

So, in version 2.0 you have the multiple inheritance, you have a more sensitive and better overloading resolution mechanism, you have type-safe linkage to make sure you can link larger programs together more effectively, and you have abstract classes. The list is fairly long but not radical. We have a lot of users, and we have to make sure there is a certain stability in the growth.

DDJ: Are you planning specific features for C++ beyond 2.0?

BS: We've been talking about exception handling and parameterized types for a long time. It's universally agreed that we need them. We have a reasonably good design for parameterized types that I presented at the last USENIX C++ conference, and that needs to be refined a little bit. Exception handling is one stage behind that, but we need it badly. When you go to C++, you get more ambitious. You want to have larger libraries, you want to use more of other people's code, and so you need more support. That's what we're trying to provide as quickly as we can without just throwing in everything at random. The language has to be kept coherent.

DDJ: The Integrated Development Environment with its integrated editor, compiler, and debugger has become the chosen software development suite in small systems. Turbo C, QuickC, Turbo Pascal, etc., are examples. Stand-alone symbolic debuggers that deal specifically with objects like C++ classes are beginning to appear as well. Are such environments appropriate for C++, and do you know of any current developments?

BS: Oh yes. They'll come. Some people will use them, and I believe you can buy one from ParcPlace now. I'll assume that since Microsoft is working on C++, they'll be working on a suitable environment. Borland is playing the same game they'll be doing it, too. I know of several other people who are thinking along those lines.

The thing that one has to seek is code portability across the different platforms. Certainly you can provide better tools for a specific platform, a better compiler, better compile times, a better program development environment. Unless, however, you are able to take your programs out of their environment and export them to something else, you have painted yourself into a corner.

DDJ: To date there is no standard for C++. Hewlett-Packard formally requested the ANSI X3J11 committee to undertake that standardization as a compatible superset of ANSI C, but the committee was ambivalent about it, failing to vote to begin the task. One reason for the request was to avoid the time-consuming overhead of setting up a new committee. Opponents to the idea pointed out that you are still in the process of the C++ definition and, as such, are not ready for standardization. Do you see the committee's action as an impediment to the acceptance of C++? Is C++ ready for standardization?

BS: Life isn't easy. Clearly we would like a fully standardized language, and equally clearly we don't know how to do that. There are still some features that we need to design. There were discussions that included me, other people at AT&T, and people from Hewlett-Packard and Microsoft. Where do we go from here, we asked? How can we get the most stable environment the fastest without freezing the language at a level where everybody has to extend it themselves? If we standardized C++ simply as it was, everybody would build their own exception handling and parameterized types.

I don't think that the ANSI C committee would be at all a suitable forum for standardizing C++. First of all, they are not C++ users. They may be C experts, but they are not C++ experts. We might as well say that since the Pascal committee did a good job on Pascal, let them do C++. We need a new commit tee composed of people with C++ experience. We clearly need a standard as soon as possible, but we simply have to figure out what "as soon as possible" means.

We need a C++ that's as compatible with ANSI C as possible, but it can't be one hundred percent compatible without destroying every C++ program ever written. When ANSI turned down the proposal, Hewlett-Packard went to SPARC, the policy committee for ANSI, with a new request to start a new ANSI C++ committee with the charter for standardizing C++, a committee composed of people who know the problems of exception handling and parameterized

types, and who know that one hundred percent compatibility with ANSI C is not desirable -- as close as possible but no closer. That was accepted by SPARC, and they have sent a recommendation to X3 to start an ANSI C++ committee. Presumably the first meeting of that will be next spring.

DDJ: "As close as possible to C but no closer." That was the title of a paper by you and Andrew Koenig wherein you identified the differences between ANSI C and C++. The tone of the paper was that those differences are proper and necessary given the different purposes of the two languages. The paper also stresses the inconsequential nature of most of the differences. Do you see those differences as proof that C and C++ can not be combined? Should they be combined?

BS: I don't think the two languages should be reconciled further than they are. The differences are quite manageable by conditional expressions. If you are writing C++, you use the new keywords and it isn't ANSI. If you are writing ANSI C you could have an option in your compiler that suppresses the C++ subset, and it will not affect the way you write code. It's much more an issue for language lawyers than it is for programmers. The whole thing can be exhaustively discussed on two pages and the differences can be listed in about ten lines.

DDJ: Will you rewrite your book, the C++ Programming Language, to reflect the new features of version 2.0?

BS: I will as soon as I get time. But it's more important to get the language stabilized, and so I'm working on a new language definition, a new manual. I am working on a book, as well, with Margaret Ellis that explains what C++ is, not, as the first book does, how you go about using it. The book states what the language is, what the implementation techniques are that make sensible implementations of certain parts, and why certain decisions were made.

This is a "what" book, not a "how to" book. In theory, it's a book for experts. If you have a question about what the language is, the answer should be there. Lots of people prefer to learn languages from such explanations. You never know who would want something like that. I learned ALGOL 60 out of the ALGOL 60 Revised Report. I'm not the only one who is sort of semi-masochistic in the way I read things.

DDJ: When will this new book be available?

BS: It's supposed to happen in December or January. The question is whether I can make it. The manual work itself is taking longer than it should. It's about twice the size of the original manual, not because version 2.0 is twice the language as version 1.0, but because I need to go into much greater depth.

If you can assume your reader's culture, you can take shortcuts in explanations, and there are certain words that you can forget to define without getting into trouble. C++, however, is breaking out of the C ghetto. People who were brought up with Pascal and wouldn't touch C with a barge pole are getting on board with C++. They come in and they try to read some of the standard C++ literature. When you get their comments, you learn what assumptions you made without explaining them. Such things must not be done in the manuals we are working on. It's getting harder to write a manual because the audience is becoming more diverse.

DDJ: One of the problems we've observed when function-oriented programmers attempt the transition to object-oriented systems is that there doesn't seem to be any way to describe the new paradigm to them in a way that they can learn it without actually using it.

BS: We'll eventually do better. The first books on C++ just said what the language was, put a thin veneer on top of it, or described it primarily as a better C. Lipmann's book and Dewhurst and Stark's book go beyond that and demonstrate how things are done. We have a major education problem on our hands. I've been saying that for years. You can write Fortran in any language, and if you only use C++ as a better C, you'll see improvements in your programs and productivity, but you won't get anywhere near what you can do or what we have seen demonstrated with greater degrees of data abstraction and object-oriented programming.

DDJ: The function-oriented programmer does not understand intuitively what the advantage of that is.

BS: It's hard to explain how to bicycle. I can talk myself hoarse and still you go up to a bicycle, and you fall off the first time. A certain amount of practice must be done. Programming is an art like riding a bicycle. It's learned by doing it. But you can certainly help the process. You don't just say to somebody, "Here's a bicycle. Ride." Similarly we need better education on how to use a language like C++ for object oriented programming. You can write better articles, use videotapes, put training wheels on the new programming environments, help programmers get started. It can be managed. It will be managed. The gains in doing so have been demonstrated often enough.

DDJ: Programmers have been told that they must unlearn what they have learned in order to use object-oriented programming, and they reject that.

BS: That's not what I am telling them. C++ is a better C, and it supports data abstraction, and it supports object-oriented programming, and, yes, you can forget all you have learned and jump into the deep end. But lots of people do it differently. They start out using it like a better C, they experiment on the side with the techniques they don't quite understand. Follow the literature so that you know the syntax, the basic semantics, and your tools before you take that big leap. Then try to focus on classes for a new project.

One of the advantages of C++ is that you can take some of the preliminary steps without a paradigm shift from function to object-oriented programming. You can learn the tools, the language, the basics, how to use debuggers, whatever it takes. And then one day when you find the right project, you can try the next step. We've seen that done quite a few times.

There are people who enthusiastically read all the literature, go to C++ conferences, and then go straight in, design huge class hierarchies, and program them. I'm always amazed when it works, but it does quite often. There are people who can write perfectly standard designed C programs, go away and wait two months and come back and write their first program in C++, truly object-oriented with tens of thousands of lines, and, lo and behold, it works. By all laws, it oughtn't. But it's happened.

Of course, I'm sure there are also people out there who have gotten burnt trying to do that.

DDJ: Would you offer your comments about the future of programming. What kinds of things do we need to understand in order to deal with software development in the near future.

BS: I think we'll see much more emphasis on the design of classes and the formal interfaces between parts, and an increase in the reuse of existing programs and libraries of classes. We will need tools that help us do this and draw structure or draw inference on what the structure is. Things like performance analysis and coverage testing will all be available. It's worth remembering that it's not a solitary activity -- not just one guy sitting there with one machine; many of the key activities are social.

We'll also need to develop ways of talking about programs that are ahead of what we are doing today. It's no good if we can compose programs and components out of classes if we can't talk about that activity in a sensible way. We don't have the vocabulary. It would be nice if we knew what object oriented design was. By and large, we don't know it yet. But that will emerge in five years and will be virtually accepted almost universally after that.