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 Bjarne.

Dennis Ritchie is the designer of the C language and is the "R" in K&R, the nickname for The C Programming Language, co-authored by Brian Kernighan. He is a member of the Computing Science Research Center at AT&T Bell Laboratories in Murray Hill, New Jersey.

Dennis did his undergraduate and graduate work in physics and applied mathematics at Harvard University. Since joining Bell Lab's Computer Science Research Center in 1968, he has worked on the design of computer languages and operating systems. Along with others at Bell Labs, Dennis created the Unix operating system, and designed and implemented the C language. His current research is concerned with the structure of operating systems.

DDJ: As the designer of the C language, you are no doubt the world's very first C programmer in a world where the number of C programmers is inestimable and growing fast. Yet, certainly C was not your first language. When did your programming career begin and with what systems?

DR: I started when I was in college in 1961. I was a physics major. There was no such thing as a Computer Sciences curriculum then. The Comp Center at Harvard offered an informal course in programming on the Univac I, and I went to the IBM office and got manuals. In graduate school in 1963 I was the teaching fellow for the introductory programming course. For a while I worked at Project MAC at MIT. My graduate work was theoretical in recursive function theory. I lost interest in that aspect of things when I finished there, and I've been spending most of my time programming ever since.

DDJ: Do you actively program now?

DR: It depends what you mean by programming. There's a fair amount of looking at stuff and deciding how it should work. These days there's more bureaucratic stuff. I'm not in management, but I write memos, look at proposals, complain to the X3J11 C committee, and things like that. I'm definitely still involved in the technical aspects of things.

DDJ: The ANSI X3J11 committee has been five-plus years in arriving at a proposed standard for the C language. How long did it take you from the time you had your original idea for a C language until you had the first compiler running?

DR: The C language grew out of an earlier language. The syntax of the early C language was essentially that of B. Over the period of a couple of years it grew into something like its current form. The most significant milestone in the growth of the language was when the Unix system was rewritten in C.

DDJ: How long did that take?

DR: Mostly it was done in the summer. There were two tries at it. This was in 1973. The summer before, Ken Thompson tried to do it, and gave up. The single thing that made the difference was the addition of structures to the language. When he first tried there were no structures. They were in by the next summer, and this provided a way of encapsulating or describing the data structures within the operating system. Without that it was too much of a mess.

DDJ: You mentioned complaining to the ANSI X3J11 C committee. What was the extent of your participation in the development of the ANSI C standard?

DR: My participation in the committee was really quite minimal. I sent them a couple of letters. One was to point out the consequences and difficulties of the path they were taking with the new style function definitions and declarations. It's clear that the new style -- function prototypes, as they call them -- is a good thing. The language is better for having it, and it should have been done that way the first time. The problem, however, is in the interval before prototypes are universally accepted, while you still have both the old and new styles. I pointed out that with that approach there will be confusion and the possibility of errors. For example, if you think that there's a prototype in scope, you might call the function and expect that the arguments are going to be coerced as they would be in regular ANSI C. But it might not happen.

DDJ: In the Rationale document, X3J11 has paved the way to eventually do away with the old style of function declarations and definitions. Will that solve the problem?

DR: Yes, but in this interval there is a sticky situation. There are complicated rules for what happens when you mix the new and old styles. They covered all the bases when they made the rules, but the rules are messy, and most people couldn't reproduce them or explain what they mean. The letter I wrote was to suggest that maybe they should think about not doing it if only because it's too late, or as an alternative they should consider requiring an ANSI compiler to have the new style only.

My second letter was related to this "no alias" business that came up about a year and a half ago. I felt more strongly about this issue because I felt they were about to make a bad mistake, and I was willing to spend a lot of time getting them to reverse it.

Around December 1987, when they were intending to produce the penultimate draft, the one that had all the technical things in it (with possibly some language polishing needed, but nothing important), something that had been simmering a long time came to the boil. Some people wanted to put in a mechanism that would reduce the problems that optimizers have with aliasing.

Here's the problem. Suppose you have a single function that has two pointers as arguments, and the function can never be sure that the pointers might not point to the same thing. Or, suppose one of the pointers points to some external place. The function cannot tell where the pointers are going to clash. According to the language rules, this kind of thing is possible, and optimizers have to be very conservative about it. In most functions it might never happen, and so the conservative compiler will generate worse code than it would otherwise. Languages such as Fortran have an easier job of this because such aliasing is simply forbidden. There's no enforcement, of course, but the compiler can take an optimistic point of view. If your program doesn't work, someone can pull out the standard and say you shouldn't have done that. Aliasing was a plausible thing for the committee to think about. It does, in fact, make C somewhat harder to optimize. The mistake they made was in trying to design a facility to allow the programmer to say that a particular function has no aliasing problem. But they actually blew it. The language rules that they developed, even after many sessions of hard work, really just weren't correct. Their specification for how you say "no alias" was broken and would have been much more dangerous than not having it. If this had happened three or four years ago, people would have

seen that this was wrong, fiddled with it, and either thrown it out or fixed it one way or the other. But this was supposed to be the next to the last draft, and all the technical requirements were supposed to be already done, and it was just broken.

That December I drafted a long and strongly worded letter to them saying that this just won't do, and pointed out the problems that I'd found. I even went to the meeting, the first X3J11 meeting I'd been to, and argued against it. What got me worried and annoyed was that this had happened when it did. If the thing had gone ahead it would have been a real bug in the specification. On the other hand, fixing it essentially meant a technical change, an important, non-editorial change, and they would need another long public review period. The point of view that I advanced was to get rid of it. I figured that my argument had to be simple to understand. If I had said, "This 'no alias' is broken, here's another thing that you should do instead," I could see us getting bogged down endlessly worrying about the technical details, so I figured it was better to argue that they should just throw it out altogether.

That was the only really detailed involvement I had with X3J11. The outcome was that there is no specification for "no alias." They voted it out. Except for some slight fiddles, the draft that is now before X3 is technically identical to what it was nearly two years ago.

Aside from those two issues, I left them alone for two reasons. One is that to take part in a standardization effort is an enormous amount of work. There are three one-week meetings a year all over the world, a lot of detailed reading, and I really didn't have the heart to do that. The second reason is that it became clear early in the proceedings that the committee was on the right track themselves. Their charter was to codify and to standardize the language as it existed. They decided in advance to do that and that is what they did. They did add some new things. The function prototypes are by far the most obvious, and there are a lot more minor things, but mainly they stuck to their charter.

I think they did a very good job, particularly when compared to the things that are happening in the Fortran committee, X3J3, where there are wide swings back and forth about the strange new things they're adding in, taking out, and putting back in. They have great political arguments between customers and vendors, Europeans versus North Americans, and it really seems to be a free-for-all. Even though some wrangling went on in the C committee, with the people involved seeming fairly fierce when you looked at it from outside, it's obvious that X3J11 was a comparatively tranquil and technically wise group.

The upshot is that I think they did a good job. Certainly, though, if I'd continued to work on things, some of the details would have been different.

DDJ: Are there any major areas where you disagree with the standard as it exists now?

DR: There are some obvious weaknesses. For example, they have never worked out what const really means. One of its intents is to say that this is some data that can be put into some read-only storage because it's never going to be modified. The definition that they have now is sufficient for that. But they also had other ideas about what it should mean, having to do with optimization, for example. The hope was that const is somehow a promise that the compiler could assume that the data item wouldn't change underfoot.

If you have a pointer to a const, one might hope that what is const is not going to suddenly change secretly. But, unfortunately, the way the rules read that's not actually true. It can change, and this wasn't just an oversight. In fact, they are potentially overloading the meaning of const. There are ideas involved other than what people hoped to get, and they never really worked out exactly which ones they wanted and which ones they didn't want. It's a little confusing.

It's generally recognized that the standardization of the library was as important as the standardization of the language. Among Unix systems there are few variations on what's available in the library. Most things are pretty much the same. In recent years, the use of C has spread far outside of Unix systems, and the libraries supplied with compilers tend to vary a lot, although many of them were based on what was available on Unix. So the standardization of the library is important. On the other hand, I've heard lots of complaints, both from users and implementors, that what they standardized and some of the rules and interfaces for library routines were not very well worked out. There may be more there than is necessary. Things got too complicated.

DDJ: What was the rationale behind the decision to leave out the read, write, open, close, and create functions?

DR: Those functions are viewed as being quite specific to the Unix system. Other operating systems might have great difficulty in supplying things that work the same way those do. The idea of the original pre-ANSI standard I/O library was to make it possible to implement those I/O routines in a variety of operating systems. It took us a couple of tries to reach that particular interface. The machines we had here were the PDP-11 running Unix, a Honeywell 6000 running GECOS, and some IBM 360s running various IBM systems. We wanted to have standard I/O routines that could be used in all the operating systems, even those that didn't have anything like Unix's read and write. The committee felt that it was better to let the IEEE and other Unix standardization groups handle that. They specifically avoided putting things in the C library that were Unix-specific unless they had meaning in other systems.

They did another thing that people don't quite understand. They explicitly laid out the name space that a standard compiler is allowed to usurp or claim. In particular, the guarantee is that there is a finite list of names that the compiler and the compiler system take up. These are simple names, beginning with underscore, and are listed in the back of the standard, something like keywords. You are allowed to use any name that isn't on this list. In an ANSI-conforming world, you are allowed to define your own routine called read or write and even run it on a Unix system. It's guaranteed that this will be your routine and that your use of the name does not conflict with any I/O that the library itself does on your behalf. The Unix library authors will be constrained to have an internal name for read that you can't see so that if you bring a C implementation from a big IBM machine or from an MS-DOS machine and you happen to use the name read for your own routine, it will still compile and run on a Unix system even though there's a system call named read. They have circumscribed the so-called name space pollution by saying that the system takes these names and no others.

DDJ: How can they be sure, that an implementor won't need other than those specific names from the list?

DR: There are rules for how the internal people can generate names, namely these underscore conventions. The end user is not allowed to use underscore names because any of these might be used internally. There is a problem, though. They made a list of things and said they'll do this and no more, and that helps. But there is still a problem for the writer of a library who wants to sell or distribute it. You're in a bind because you don't know all the underscore names that all the implementations are going to use. If you have your own internal names, you can't be sure that they're not going to conflict somewhere. If they have underscores, they might conflict with the underlying implementation. If they don't, then they might conflict with things that your end users are going to use. The C committee did not solve the problem that other languages have tackled explicitly. There are other ways of controlling the name space problem. They made a convention that helps, but it certainly didn't solve the real problem. It solved it enough to improve the situation. The basic problem with an uncontrolled name space is that if you write a program, and it just uses some name that you made up, it may be actually difficult to find out that this is not the same name as some random routine that's used internally by your system

library. Unfortunately, we here at Bell Labs are in a bad position to notice this and do something about it because in our group we've simultaneously developed the compiler and the library and the Unix system, and so people here tend to know the names.

So, to summarize X3J11, the two largest things the committee did were function prototypes and the standardization of the library. It was more work than anybody expected, but I'm perfectly happy with what they did. The only problem was it took twice as long as they thought.

DDJ: Do you see a potential for other standard extensions to C, beyond those added by X3J11?

DR: One of the major excuses they give for not doing something is that there's no practice, no prior art. So obviously people will try to create prior art for the things they'd like to have happen. One such group is the Numerical C Extensions Group.

There are people who have strong views about what should happen. The idea is to get together with this group and agree that these are the things we need to have, so let's make some rules so that when people try things out, we'll all be trying it the same way, and we'll have a coherent story to tell, if there are going to be these extensions.

Most of them have to do with IEEE arithmetic issues, exceptions and such, for example. There's a core of things that are more general, and one that interests me is variable arrays with adjustable sizes. One of the things C does successfully is deal with single-dimension arrays that can be variable in size, but it doesn't deal with multi-dimensioned arrays that are variable at all. This is an important lack for numeric types, because it makes it hard to write library routines that manipulate arrays. Multiplying two arrays is a bit painful in C if the arrays are variable in size. You can do it but you have to program it in detail and the interface doesn't look pleasant. That's an obvious need, and I volunteered to look at how it might be done.

The NCEG will probably try to become official. They will affiliate themselves either with X3J11 or as an IEEE standardization organization. This would give them more clout. Also, many of the companies involved worry about legal issues. Companies who are members of informal groups deciding standards worry about anti-trust, whereas if they are members of official, blessed standards organizations, then they can contribute. They worry about being accused of going off into a corner and doing things behind other people's backs. It's better to do it in the open. This may be just some lawyer's nightmare. NCEG will probably become a subcommittee of X3J11.

DDJ: One non-ANSI extension to C is C++, a superset language that surrounds C with disciplines and paradigms that go beyond its original intent as a procedural language. Can you comment on how appropriate that is and how successful it has been?

DR: Let me confess at the start that I know less about C++ than I probably should. C is a very low-level language on a variety of fronts. The kinds of operations that it performs are quite basic. The control over names and visibility is basic. The defects or limitations of C in this area are most evident when you get into a large project where you need strong standards, rules, and mechanisms outside the language. Language developments such as C++ are trying to supply some of the structure within the rules of the language for controlled visibility of name space and are trying to encourage various kinds of modularization. This is good, I suppose.

C was designed in an environment where modularity was encouraged not so much by the language but by the kinds of programs we wrote. In the Unix system, the tradition is for small utilities that work together as tools, and the interfaces between them were set by the conventions and rules of the operating system, i.e., pipelines and so forth. The complexity of the pieces was kept low by custom. Commands tend to be simple. In the world today, there's a certain amount of admiration for that point of view. Certainly the appreciation for that style is

part of the reason for the growth of Unix. People now are undertaking the building of much bigger systems, and things that we handled by convention ten or fifteen years ago must be handled by more explicit means. C++ is one such attempt.

Bjarne decided to design a compatible superset of C and to translate the C++ language into C code. That approach is not without its problems. First, having decided that C++ is going to be largely compatible with C, every time he departs from that he's under pressure either because of some accident or because ANSI changed something. Or because he feels that there's something he has to differ in, people are going to complain and get confused. Second, he is constrained by the choice to make a C++ to C translator possible, that is, he is constrained, as C was, by the existing tools of the various systems. The whole separate compilation business in C++ is made a lot harder by the desire to make it work with existing tools. If he could have simply designed a language and implemented it, then a lot of the anguish would have been avoided.

DDJ: Rumor is that within Bell Labs, C++ is now called C, and C is called "old C." Any truth in that?

DR: I've asked Bjarne not to say "old C," and, as far as I know, he has complied with that request.

DDJ: Colleges and universities have started offering courses in C. Some C tutors have observed that many instructors either don't understand C well enough or they don't understand teaching well enough to insulate the novice student from the kinds of things you can do in C, things that the student cannot grasp. In light of that, and as compared to Pascal, how do you view C as a potential teaching language?

DR: Obviously, C was never designed to be a teaching language. It was designed as a tool to express the kind of programs that we were trying to write at the time. And it's fairly low level in that concepts, like pointers, have a prominent role. I would not argue that C is a particularly good language for teaching programming. As Pascal was explicitly designed for that.

Pascal's main fault is that you cannot use Pascal originally designed to express all the things you need to, certainly not in a systems environment, and not for general applications either because of explicit constraints that are built into the language. C was, from the very start, designed to do all the things that we found necessary in order to express ourselves, and little design thought was given to preventing people from using its powerful features.

Nevertheless, it's possible to teach C in a way that's reasonably safe if you start with parts of the language that are similar to other procedural languages. Then you can teach C's more unusual aspects -- pointers, for example -- as cliches or set ways of expressing array manipulations and so forth. Later you can gradually widen out into the more general things possible with pointer manipulations.

I have not had the experience that the tutors have had. Part of the difficulty with being in a position like this is that you have very little opportunity to see what the novice really feels. But perhaps the reason there are not better instructors is that things have grown fast, and there might be people teaching C who only recently took the introductory course on the language themselves.

DDJ: Would you attempt a prediction for the future of the C language?

DR: I think the period of C's largest growth is over, although it will be increasingly used and it probably will not change very fast. The new language developments based on C will be on

successors such as C++ or perhaps some things we haven't heard of. In terms of what C tried to do, I think it succeeded fairly well. The goals were reasonably modest. There's still plenty of work to be done finding languages that have the touch of reality that C has, work where you handle real problems in real environments as opposed to dealing with elegant creations that can't be used. Sometimes things can't be used just because the compilers don't exist on the machines people have. Sometimes it's because there are simply flaws in the design, not from the language point of view, but from the point of view of what the language ends up doing in the real world. And in that respect, C seems to have worn fairly well.