Oral History of Bjarne Stroustrup

Interviewed by:
Paul McJones

Recorded: February 5, 2015
New York, New York

CHM Reference number: X7399.2015

Paul McJones: Today is February 5, 2015. I'm Paul McJones in California.

Bjarne Stroustrup: I'm Bjarne Stroustrup in New York City.

McJones: And behind the camera is Jon Plutte. How are you Bjarne?

Stroustrup: I'm fine, thanks. Thanks for taking the effort. It's amazing we can sit chatting like this[1].

McJones: Yes, several thousand miles apart. And we're probably using some C++ technology behind the scenes.

Stroustrup: Definitely. I was thinking that when I started on C++ we couldn't have done this.

McJones: Right. I thought it would be good to start at the beginning, if you could tell us a little bit about your parents and where you grew up.

Stroustrup: I was born in Aarhus in Denmark and I lived there for the first 23 years of my life. I lived in an apartment in a working class district. My father was somebody who laid down floors and late in life he was a hospital porter. My mother was a secretary. She had a middle school education which was the most anybody in the family had had by then. I went to the local school, which wasn't a great school. Actually, I've seen the statistics; it was the worst school in town but somehow I managed to get into high school and from there to the local university which happens to be a good one. The University of Aarhus appears on the top 100 universities in the world ever so frequently, not every year but lots of years. I actually had a pretty nice time in Aarhus growing up and a very stable situation. We lived in one place known as "the Yard." One of those standard rectangular apartment buildings that you have in Europe with a yard in the middle with some grass and such. A very nice place. I've traveled far and lived in other places but it's not because I disliked where I came from. I've still got family in Denmark.

I started mathematics with computer science in the university and that's curious because like most high school students I hadn't a clue what I wanted to do. I was looking around and I wanted to be a historian, an architect, a sociologist, an engineer and then decided I that I really didn't know what I wanted. So I just got grades that were sufficient to get into anywhere. I was thinking of going to the engineering university in Copenhagen but decided that the costs were too high. In Denmark, of course, you have free tuition so that's not a problem, but coming from a family with few resources the idea of taking on debts for a six-

[1] The Skype video chat application was used for the interview.

year course living in a big bad city, meaning Copenhagen, was just too much, so I decided I'd do something local. Since we [Aarhus] didn't have an engineering school and since I had been reasonably good at math in high school I signed up for math. But I was convinced that I needed to do some practical math. I wanted to actually practice something, to do something. I had an absolute dread of becoming a teacher because I thought that was what happens to bright kids who have no role models, have no ideas what they're doing. I didn't want to be a teacher. I needed to find some kind of applied math, something I could really do. And so I signed up for computer science. The word computer is not in the title of that in Danish ["Datalogi"], so I was quite capable of misunderstanding it and thinking it was math. In my second year, I realized that I had been wrong and that programming, data structures, machine architectures are not math, at least not at that level. This was pretty good for me because it turned out that like most other people I wasn't quite as good at math as I thought I was. On the other hand, programming was really fun and so were learning about machines and computer science in general. I haven't looked back since.

McJones: Were there any mentors or anyone who had influenced you to go down this path up to this point?

Stroustrup: Not in any sort of real sense. I've thought about this problem before and I did have a math teacher for nine years running who was a very good math teacher, so that's probably why I didn't flunk math and I actually was reasonably good at it. But he himself had been trained as a Danish [language] teacher and had never been to a high school. He had gone from middle school to a teacher's training college and he couldn't guide me beyond that. Nobody in my family could, so once I was in high school I was sort of on my own. With a bit of luck I ended up in that computer science program and got my master's in mathematics with computer science. I never regretted the amount of pure math I ended up doing. You never know what comes in handy in your career. If you'd told me, say twenty years ago, that algebra would have turned out to be really useful I would probably have argued strenuously against that idea, but with type theory, Alex's work on the STL [Standard Template Library], that kind of stuff, the discipline of thinking that I learned in algebra has actually been very useful over the last couple of decades after being not useful at all for a couple of decades.

McJones: Right. But it was latent in there.

Stroustrup: It was there.

McJones: Do you remember particular textbooks that seemed important to you at the time?

Stroustrup: What pops into my head is a book on the open game of chess. I don't think that counts. Gries's book on how to write a compiler[2]—very practical. The only problem was it was mostly grammar

[2] David Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, Inc. New York, 1971.

and parsing. When it came to real design that was the least important part of the compiler, but it got me started and interested in languages and how to implement languages. I was never a languages guy, though. I was interested in operating systems, I dealt with microprogramming a lot, machine architecture. One of my early projects in Aarhus was to design a virtual machine in hardware for BCPL. I needed to run BCPL or rather I wanted to run BCPL programs, so I looked at OCODE,[3] which is basically assembly code for BCPL, and thought, "Why I don't just implement that in hardware?" So I wrote a little piece of microcode that implemented it instruction for instruction. I made a binary representation of OCODE with compressed data structures so that you could have a compact program and then I could run BCPL. This is not quite the traditional way of implementing a programming language, but Martin Richards had kindly provided this intermediate step in the compiler which I could hijack. I learned probably a dozen, maybe twenty languages before I left Aarhus. I got a master's [Cand. Scient.]. That's what you got in those days. There were no bachelor degrees. . So I was not a languages guy, but I did know, say, twenty languages.

McJones: I was just curious, having learned that many different languages, did that include practical ones and also some of the research languages?

Stroustrup: The distinction between practical and research languages was never explained to me and I don't think I appreciated it. The first course, I think, had three or four languages. I remember writing some COBOL and hating it, writing in lots of languages. Some of them I liked and some of them I didn't like. When I say 20 languages it doesn't mean I have completed a major project in each, but it means I had written a couple of small programs in each [plus major projects in a few]. You appreciate what a language can do and what it cannot do. What is that string manipulation pattern matching language from the sixties?

McJones: SNOBOL?

Stroustrup: SNOBOL. That's right. I remember finding SNOBOL really interesting and useful. What I'll call a formative experience was realizing that I had written an ALGOL program in SNOBOL and that it was a really stupid program. It was complicated. It was difficult to get to run. It ran slowly and it was not how any sane SNOBOL programmer would write it. So I had to go back and learn the idioms. You can't just take a language and start writing in it as if it was your previous language, which is what we essentially all do at first. Then we realize that we're doing it again and correct. That I actually learned, I think, in the first half year of programming. I picked up quite a few programming languages. In the old days, when languages were simpler, programming environments were less complicated, and libraries were less extensive, you could actually find yourself sitting, waiting for a couple of hours somewhere, read a language manual to pass the time, and then go back and write some programs when you got out of the waiting room or wherever it was you were stuck. Then you could play with it for a couple of weeks or a

[3] Martin Richards. "The portability of the BCPL compiler". Software Practice & Experience, 1: 135–146. doi: 10.1002/spe.438010204

couple of months, do something it was suitable for and finally get on with something else. But languages were never my main concern. Languages were tools for doing something like getting hardware to do the right thing or getting hardware to have the right instruction set or something like that.

McJones: In Steven Lohr's book *Go To*[4] he mentioned that you were a contract programmer at that time. Did that have a strong influence on you?

Stroustrup: It did. It wasn't meant to. It was just meant to give me some cash so that I could avoid taking on debt and live frugally, but reasonably, up to my master's degree. Some of the microprogramming, again, was done on contract from the university. But my job from Burroughs[5] was actually interesting because there I got to see the process from negotiating with the client what should be done, to deciding what could be done—out of what the salesmen wanted and what the client wanted—, and then building a demo. I had a very simple business model. If I could build a demo that the client was willing to pay for then I got the job of implementing the system. Now from the corporation's point of view this is safe because if I sell a demo I cannot make real I wouldn't be working for them anymore. On the other hand, it is very much in their interest and the client's interest and my interest to get the system as nice as possible. So I was out talking to clients, I was out talking to them when they were installing the system, seeing how it could install, see how to train the people who are actually going to use it, sit around chatting with the client's children realizing that if I screwed up, these children will get hurt. As I went along, I did this many times, between six and a dozen times over a few years and then I thought about it later. I had actually seen the whole process of software development, from trying to figure out what system might get built, to doing the clean up once something broke, and everything in between. I have fairly strong opinions about systems being appropriate and reliable and affordable which, I think, partly came out of that. It is not something that is easily instilled in a classroom setting.

McJones: Right. It reinforced those feelings you had had initially that you wanted to go into an applied field, but it really made it concrete.

Stroustrup: Doing something practical and sitting there on the sofa trying to explain to a couple of kids what I'm doing for their dad was an interesting exercise.

McJones: So at that point you could have easily started some sort of a software development career, but instead, you decided to go to graduate school. What led to that?

Stroustrup: I actually got to England before I got my master's because I was working with some microprogrammable hardware and they had some really cool stuff in England. So I went over there for a

[4] Steve Lohr. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists, and Iconoclasts—the Programmers Who Created the Software Revolution*. Basic Books, 2002.
[5] Burroughs Corporation, now part of Unisys Corporation.

while. While over there, I got the idea that it actually was interesting and I could do something more. Getting a Ph.D. was my solution to the problem of how to spend more time in England and play with some more interesting hardware. So I applied to the English universities I had heard of and actually managed to get accepted into most of them. I was almost going to Newcastle but my then girlfriend [now my wife] explained to me that if you have an offer from Cambridge you take it. This is nonnegotiable if you are in England. I haven't regretted that. It turned out to be a good choice. It's probably still the best place for practical computer science in Europe.

McJones: It certainly seems to have to turned out well for you. Your advisor was David Wheeler and his advisor was Maurice Wilkes and these were the people who got the first practical computer running, so it really connected you to the origins.

Stroustrup: Maurice Wilkes invented microprogramming. Would you believe it, I sat in an interview with Roger Needham and Maurice Wilkes at Cambridge and it somehow hadn't dawned me that Wilkes was the one who invented microprogramming. And so I was explaining all of the cool stuff about that without knowing who I was describing it to, but I must have done a good job because I got accepted. That was the toughest interview I've ever sat through. There were these two guys. Wilkes got the second Turing Award, so he's no slouch, and a Computer Museum History Fellow, by the way. I'm a third generation here because later David Wheeler got a Fellowship and now I'm getting it. And there was Roger Needham, who later became vice chancellor of the university and started the Microsoft [Research Laboratory] in Europe there [in Cambridge]. They were taking turns. One of them asked a question, the other one listened. When we run out of that question the other asked another question. I don't think I've had as hard three-quarters of an hour ever as that interview. Halfway through they explained to me that I was all wrong [about my suggested area of research] and was there anything else I could do? So I had to change horses in the middle of the interview but that was interesting. I remember it fairly well.

McJones: That sounds pretty tense. But then you probably got to know them a lot better, say two years later how did you look at them?

Stroustrup: The teaching style at Cambridge was somewhat different than most people think. I had a master's when I came in so I didn't actually need to take any courses. I took courses when I needed them. David Wheeler's teaching style was very simple: you came to him when you thought it was a good idea and I quickly found that turning up about once a week was a good frequency. You come in, a new grad student with your head full of great ideas, right. You come in and explain these great ideas to David Wheeler. And he sits there and says thoughtfully, "Yes, Bjarne, that's not a bad idea. You know, we almost did that for that EDSAC II," which was the most dreaded words because that meant that in about 1956, about the time I entered primary school, they have thought of it. Then he could go on for about an hour explaining what they did instead, why they did it instead, what tradeoffs were,why what they're going to do instead didn't quite work, and how they fixed it, again and again. So basically his tuition style was to listen to what I said, then shoot it full of holes ever so politely, and then giving an impromptu lecture for

about an hour on topics related to it. I then went away feeling about this big [showing about a couple of inches with his hand]. Next week, you come back for more of the same.

I also talked a lot with Roger Needham, who was one of the senior lecturers at Cambridge [A senior lecturer then in Cambridge is what we'd call a full professor in the US today.]. Wheeler and Needham both got promoted professor when Wilkes retired to go to the U.S. to become a consultant for a decade or so. But anyway he [Rodger Needham] was more chummy with the students. We used to go to lunch with a group of students where he amazingly was able to drink two pints of Abbott, which none of us would have been able to do and still work in the afternoon. His style was different. You want to talk to him, talk about something specific and he walks up and down talking. This is more two-way than it was with Wheeler but he walked up and down. I developed a technique standing roughly on a corner and seeing him walk like that <gestures> because otherwise I would get really tired just to keep up with him and he had lots and lots of good and interesting ideas. Most of what I learned from Cambridge was from talking to these two guys, that way.

I didn't see much of Maurice Wilkes while I was at Cambridge. He was the remote lead professor sitting in his office. You had to book a time with the secretary and if you were a lowly grad student you probably were busy most of the time. I got to know him reasonably well later after I graduated. I hosted him at Bell Labs and met him at conferences. Really, really great guy. He managed to outlive his own autobiography by about twelve years.

But one of the things that Cambridge could do and later Bell Labs could do is somehow raise people's expectations of themselves. Raise the level that's considered acceptable. You walk in and you see what people are doing, you see how people are doing it, you see how apparently easily they do it, you see how nice they are while doing it, and you realize, "I better sharpen up my game." You have to get better because what is acceptable has changed. Some organizations can do that but most can't to that extent. I've just been very, very lucky to be in a couple of places that actually can increase your level of ambition and the level of what is a good standard.

McJones: So this is the environment where you pick a research topic and that is distributed computing. It's not programming languages, but you have an experience carrying out that research that seems to plant a seed for you.

Stroustrup: Yes. One of the first things that happened to me, probably the first day at Cambridge, I was sitting down in Wheeler's office and he asked me, "You know what's the difference between a master's and a Ph.D.?" I said no. He said, "If I have to tell you what to do, it's a master's." In other words, it was my job to figure out what to do. That was the first and most important job of a Ph.D. and it took me a year. The next thing was that he had forgotten that I had a master's so he explained that a fate worse than death was to get a master's from Cambridge. I was able to sit there waiting until he finished and said well,

I don't need a master's, I've got one. So that was good. But yes, I had to find the area. The distributed systems came because of my work with machine architectures and microprogramming. I was trying to develop a high-level architectural support for modularity, for having hardware-protected entities talk to each other. Cambridge had the CAP machine designed by Wilkes, Wheeler, and Needham, which had hardware protection capabilities. I wanted to extend this so that it could be on several processors in a single machine and even better, if I could, on several computers over our network. The idea was to build hardware that could do that efficiently. Unfortunately, I couldn't afford a multi-core machine-there was hardly any of them on earth at the time-and I certainly couldn't afford a network. They built the first local area network in Cambridge [The Cambridge Ring] while I was there but when I started there wasn't such a thing. So I simulated my "system" instead. I was still focused on the systems' issues, on the hardware issues, system software issues, and I got interested in modularity, how to organize systems into parts. I had picked up the knowledge of how to organize code in a modular way from Simula. I had learned Simula in Aarhus, mostly from talking to Kristen Nygaard. I was lucky to have had many, many discussions with Kristen Nygaard. You don't actually have a discussion with Kristen Nygaard. You say something and he talks to you for a while. He was a great guy and I learned a lot.

I picked up Simula and wrote a simulator that brought the university mainframe to its knees before I could get any decent data out of it. The resource demands of that simulator were just too much for the mainframe. Later I came to realize that this is typical for simulations. You just run a simulation with more data or with more entities or for a longer time until you run out of the memory and CPU power available. So this is fine. But when you do this as a grad student, you only do it once before you're kicked off the machine because the astrophysicists and the chemists have much better funding and political power. So I had to do something else. What I did was to move to the CAP, which was the experimental hardware-capability machine—a very unusual machine, strange architecture, and beautiful. You could program it in languages nobody's ever heard of and that kept the chemists, the astrophysicists, and most of the computer scientists away. I had a choice between ALGOL 68C, which is ALGOL 68 without garbage collection, and BCPL. I did a few experiments. I picked up both of those languages along the way and I was actually pretty good at BCPL at the time, having implemented it in hardware [in Aarhus]. I rewrote my simulator from Simula into BCPL and all of the high-level structure disappeared. All of the nice organization that had helped me debug and helped me design my simulator disappeared. But the resulting BCPL, once I had debugged it and lost half of my hair in the process, ran really fast. It could use all of the resources of the machine. It could communicate with anything on the machine so I got my data and I got my Ph.D. I came away with the opinion that I would never again want to attack a problem with tools that were fundamentally unsuitable. In particular, I don't want to make the choice between elegant, which Simula was for this problem, and efficient, which BCPL was. I want both. That has been one of my guiding lights. If you give people the choice of writing good code or fast code there's something wrong. Good code should be fast. I came away from Cambridge with that experience, still thinking I was going to be able to build operating systems and design computers.

McJones: So after graduating did you look at a lot of places? You weren't interested in teaching—was that true?

Stroustrup: If anything I was more convinced that I didn't want to teach. I saw it as a trap for bright kids. Also, I had developed the idea that there's something fundamentally wrong if the teacher's previous job is student. It's incestuous. There has to be some real world in there somewhere and often there isn't for teachers and professors. That weakens what is being taught and it weakens the credibility of the teacher so the effectiveness of transmission of information to students is also worse. So no, I really didn't want to teach. I looked for jobs in Denmark. Denmark is one of the nicest places on earth and if I could have found a good job there I would have gone back. But I noticed that I had made a fundamental mistake because at that time in Denmark the opinion was that if something was right and good it was being done in Denmark. Since what I was doing wasn't done it must be wrong somehow. Furthermore, for university jobs, the jobs went to people who were sitting around doing the usual thing for a couple of years until they could get a job--practicing teaching, doing fashionable stuff. What I was doing was not fashionable in Denmark. I did get an offer to write some COBOL somewhere; it could very easily have become a non-career. It didn't attract me. I got an offer to go and do something for some branch of the government, which seemed to involve counting submarines in the sound outside of Copenhagen. That didn't sound attractive at all. But there was a connection between the lab at Bell Labs and the [Computer] Lab in Cambridge. Somebody had been around and in the Eagle, where you went for a pint after a day's work. Somebody—either Steve Bourne[6] or Sandy Fraser[7]—said, "You know, if you need a job give us a call." They worked in the Computer Science Research Center at Bell Labs, so I called. Sandy Fraser responded, "Yes, you can come around. We can't promise to pay your fare but come around, give a talk, let's talk." So I flew over to the States. I also arranged a couple of other interviews, but I came to New Jersey where I was met by Sandy Fraser. He said, "Sit down." I said, what do you mean? He said, "Sit down". So I sat down and he says, "You've come at a bad time. We don't have any jobs." This is not how you want a job interview to start after you've just crossed the Atlantic, you're seriously jetlagged, and your bank account is rather low. So we didn't go to the Computer Science Research Center. We went to one of a more development-oriented place and I gave my talk. It must have been a good talk because they changed their minds and I was put straight back in the car and did a full interview in the research center and a week later I got an offer. After that, I worked there [in the Computer Science Research Center] for sixteen years, as long as AT&T still had Bell Labs.

McJones: Is it fair to say that you started off doing work somewhat similar to your actual Ph.D. distributed systems work?

Stroustrup: Oh yes.

McJones: And then that led to some new things.

[6] Stephen Richard "Steve" Bourne.
[7] Alexander G. "Sandy" Fraser.

Stroustrup: Yes. I thought now I'm in an industrial research lab; we have many more people, we have many more resources. Now, I can actually build the system I was thinking about. We had Unix and in the early days, all of Unix was right there. Unix was a reasonable size and there were people who knew everything about it like Ken Thompson and several others. Dennis Ritchie was there to show how to use C, Brian Kernighan, Doug McIlroy[8], Greg Chesson, lots of good people. So I decided I was going to build a system based on Unix by taking parts of the software and run them in different sort of units, modules, parts and these—let's call them modules—should be able to run either on a shared-program computer or over our network. We could afford a shared multi-processor. We didn't have one yet but we could in principle. And soon we were going to get an even better network. We were all sharing a PDP 11/70 at the time but there were more machines coming and it was obvious that soon a distributed-system project would become feasible. I set off trying to figure out how to separate the various services of the [Unix] kernel into separate modules and to think about how users [programs] could talk to each other through this kind of stuff.

After a while, I don't quite remember how long, probably a couple of months, I realized I was making the mistake that I had been very determined not to make, because C, which was the language of Unix, didn't have the facilities for saying which parts are separate and didn't have the ability to say what is shared and what isn't. All program parts can poke into all of the memory. Furthermore, C is so low-level that it's really hard to work with at a conceptual level. So I looked out for something better. I knew that to build the kind of distributed system I wanted, I needed language that could deal with hardware, do it well, do it efficiently, do it directly. C was a good example of that. I might have chosen ALGOL 68 if that had been available, but I soon realized that nobody was using that. It might be elegant, it might be beautiful, but this is not what was available and I had Dennis Ritchie down the corridor and Brian Kernighan. So if I really want to know how to do something for real in C, I know where to go. But C still couldn't do the high level. It didn't have the notion of a module, a separate part, an entity, well, later we called them objects. I knew Simula; I knew Simula quite well. I'd used it, I had been taught Simula by Kristen Nygaard, who was the ultimate master of this game. He invented object-oriented programming. He invented inheritance, and more. So I decided of those languages that could modularity, Simula was probably the best. There were many languages that could do the low-level stuff and C was obviously among the best. So I wanted to put the two aspects together. You couldn't put the low-level access into Simula. That was just too complicated, the compiler was too expensive, too hard to port, and too hard to target to new hardware. So I put the abstraction mechanisms into C. That's the fundamental idea of what was in "C with Classes" and now C++. There's two parts to it: the ability to manipulate hardware directly and the ability to abstract from it so that you don't have to work close to the hardware all of the time.

McJones: It seems very simple now, but yes, it was a really good idea. So it was a tool for your own project, but you talked to other people looking over your shoulder so to speak?

[8] Malcolm Douglas "Doug" McIlroy.

Stroustrup: I talked to a lot of people. And I listened, I think, reasonably well. Many of my detailed ideas, not the fundamental but the detailed ideas, were sort of wrong and I learned from people what they were trying to do, how they would like to do it, and a lot of "stuff" came into C++ from those applications. Thinking back, there were a couple of decisions I made very early on that were really important. I put in what is now known as "function prototypes" [in C and "function declarations" in C++] because I couldn't live without the ability to have a proper interface, to check your function arguments, to get conversions. The idea of sqrt(2) not being a valid program was just not acceptable, that can't happen[9]. I was complaining a bit about that, about C. Sandy Fraser said, "Why don't you just fix it?" So I did.

The other thing from my notes, one of my very first notes was that you needed a constructor to construct an object, to acquire the resources necessary to function (to establish an invariant, to use a modern term), and a destructor to clean up the mess when you're finished. This was very, very early [in the development of C++, 1979]. It's the same kind of vintage as the function declarations. There was one more thing that came in very early. It was very easy in the context of C, but this is where C++ breaks with many other languages: I really didn't want to have a distinction between the user-defined types that are created with new[10] (and live on the free store) and the built-in types that you just declare and use (and don't have to live on the free store). Simula had the distinction between what kind of types had pointer semantics (reference semantics) and what had value semantics. I was quite willing to believe that there were types that should have pointer semantics and types that should have value semantics, but I really wanted both and I really, really wanted the value semantics for user-defined types [which Simula and most of its successor languages do not have]. I never could understand why an integer was something you could have as a named variable, but a complex number, because the complex number was user defined, was something you had to put on the free store and allocate with new. I mean these two types are fundamentally the same thing. To quote Doug McIlroy from about that time, "There's nothing abstract about a complex number. It's as real as real." So from very early on, the first couple of weeks or months, you got functions with interface checking, classes with constructors and destructors, and direct support for value semantics. That's the rock bottom where things came from.

McJones: Back in the early 1970s there was a lot of research on extensible programming languages and abstract data types and yet people seemed somehow to have been distracted by inheritance and the reference-oriented implementation models and it seems that only you followed that other path and gave us the ability to define a type like the ones that were built in.

Stroustrup: Yes. It was on my list of things that I didn't particularly like about Simula and I talked to Kristen about it and he said, "Well, it's just the way it is. We have to put the user-defined types over there." [paraphrased; he was thinking about class hierarchies.] I concluded, that's not ideal, but you couldn't build the garbage collector if you didn't have that. Similarly I asked Ole-Johan Dahl, "Why don't

[9] At that time, the C compiler did not check the types of actual arguments against those of formal arguments.
[10] In C++, the new operator allocates free (heap) storage.

you have multiple inheritance?" Well, he said, "I couldn't build an efficient garbage collector if I had multiple inheritance." In fact, by modern standards he couldn't build an efficient garbage collector at all [nobody could then]. But when I was dealing with things at the lowest levels of the system I wanted the efficiency, so I couldn't afford a garbage collector. Putting things on the free store is something that is potentially expensive but destructors take care of that, minimizing the cost. But I also wanted the types I really think about, complex numbers, points, sockets, matrices, to be on the stack because that's where I have the best control, the most automatic management of resources, well, and the best efficiency. It's the ease of use and efficiency coming together, again. I wanted the important user-defined types there [as named variables on the stack]. That was something I more or less knew by the time I left Cambridge, not in the sense that I was going to do it but it in the sense that it had bothered me that I couldn't do it. C, of course, has the value semantics at the base of its object model and that helped.

McJones: In a sense, it's as if you've taken ALGOL 60, which was stack oriented, and enriched it with the ability to use the heap storage, but in this controlled way so it's very, very nice.

Stroustrup: I could have done that; I wanted to do it to ALGOL 68 but decided that would be a really dumb idea because nobody liked ALGOL 68. And if you wanted to do something practical, if you wanted to have an impact in the world, which I did, you have to do something that other people can understand.

McJones: Right. You have to engage with the currents that are there. So it sounds like at this point you're designing a language. You're also working on your original research project, but it's sometime in the early 1980s when your focus really goes more to the language side, is that right?

Stroustrup: Yes. I had my first real user after six months [Sandy Fraser, and soon others]. C with Classes was a preprocessor and it was fairly easy to get started and it was aimed at writing some initial simulations that I needed to figure out how to design my stuff [that distributed system]. Then other people started using "C with Classes". When you have users, you have responsibilities and you have to make sure that their projects succeed. As I got more and more users, I spent more and more time making sure that my users were happy. At that point I was doing the documentation, the tutorial, the fighting of bush fires, the teaching, the starting of projects, all of that. I never got to build my system, otherwise we might have had Unix clusters and Unix multi-cores about 1985, but I got totally distracted and for the next ten years I was doing this combination of language design and support of users. I mean, I was the help desk for a long time.

McJones: Yes. And I guess many of the other systems there— Unix and C and so on—had a lot of those same kinds of documentation and so on but that was typically spread over multiple people. So you were a one-man band, as it were.

Stroustrup: Yes.

McJones: So C++ emerges as a name and as a new implementation and as a more sophisticated language around this time. First going to the outside world around 1985, is that right?

Stroustrup: We had an educational release and specific releases to people who asked to use it for educational purposes so those people were using it certainly in 1984 and into 1985. In October 1985, we had a commercial release, which was known as “the brown bag release” [I still have my brown bag somewhere] because it was so cheap and because it had been so economical to build. I wrote 98 percent of everything there: documentation, build procedures, testing, language, the works. It was so cheap because I was doing research. Actually, what I was really doing was supporting users doing interesting projects. So from a corporate point of view it was free. The cost of getting a C++ implementation for a university was $75 [including compiler source code], which was the cost of putting it on a magnetic tape and shipping it physically.

McJones: Right. So essentially that was free for academic use for all practical purposes. And how about commercially?

Stroustrup: There was a rather large fee; I think it was $40,000.

McJones: Okay. But still pretty reasonable for this.

Stroustrup: Then you got a source license and you could do interesting things with it, and people did.

McJones: Were people running it only on the same instruction set that you were. Was that still PDP-11 or had you gone to VAX?

Stroustrup: Oh, no! Quite early on I decided that running on a single machine was, well, suboptimal. The people I dealt with, the people who were my users or would like to be my users, and myself, didn’t just use one machine. So my C++ compiler, Cfront, was a natural cross compiler. The way that tended to work was that I would generate C for the target machine. So if you had a C compiler we could highjack the C code generation and tool infrastructure and debuggers and such by generating C. But since C is not 100 percent portable you have to figure out how to do this. The way I did it I was to write a little program [a C program] that you run on your target machine and it figures out what is machine dependent on that system. So if you have a Unix and C, I can run this program and it will write a file that tells me what C is like on that machine: what is the size of an integer? What is the size of a pointer? Do you have signed or unsigned characters? and so on. I get a little file, which I mail back to the original machine [where C++ already works] and you just plug it in. There was a +x option that took the machine description file as an argument. The machine description file changes the compiler’s view of what it’s generating for from its own machine to the target machine. On really weird architectures like the Cray’s and the Univac it took a

whole month to port Cfront, but most of the time we got it down to about three hours, which before shipping and packaging and large tools were actually pretty good.

McJones: When this 1.0 release came out, did you actually include the binaries for a bunch of those ports? Or how did that work?

Stroustrup: I don't remember. Obviously, there was a binary for at least one machine on that tape, otherwise it would never have worked but I don't remember. We did not generate 50 binary versions and put them on the tape. Fifty is a random number.

McJones: Right. But between what you could do and what the customers could do…

Stroustrup: I know the number of target architectures was more than twenty and I didn't do all of those.

McJones: Perhaps without getting into every detail it might make sense to talk a little bit about how the language evolved. When you described C with Classes, it had inheritance but no virtual functions. It had this nice support for constructors and destructors and treating a user-defined object as a first class citizen. And now new things are happening in the language.

Stroustrup: Yes. When we changed from C with Classes to C++, which was in the winter of 1983-1984, I had been working with operator overloading. I really needed to get the value semantics, I really needed assignments, and some of my friends needed operators. They actually needed "and" and "or" and "xor" for some machine simulation, not "plus" or "minus". That came later. Actually, all the operators were thrown in the same time but the demand was not for "plus" and "minus". The really interesting operators were subscript [[]] and application [()], from which we get the current containers and the function objects. That came in then [1983]. The other thing that came in was virtual functions, which I knew from Simula, but I had been totally incapable of convincing people they were useful. The attitude was, "You want to a call a function and you don't know which one it is? Haven't you analyzed your problem? If you really need an indirection, why don't you just use pointer to a function?" I said, "Well, you can't manage that at scale and I can't teach it." So we got class hierarchies à la Simula but with multiple inheritance, because I thought I needed it. These days everybody use multiple inheritance. They use multiple inheritance of interfaces. Everybody who has a language that can specify an interface has multiple inheritance of interfaces. I hear that Java 8 has finally gotten multiple inheritance of data also. It's not as useful as interfaces, but sometimes it's just the right tool. I guess the main reason I'm mentioning this is that I've had so many years of being told that multiple inheritance was a design error so it's interesting to see other major languages adopting it in some forms.

McJones: Right.

Stroustrup: And there's something important here.

McJones: References? [I missed that question – references came into C++ in 1983 to support operator overloading – BS]

Stroustrup: Once we had shipped release 1 and release 1.2, I actually got some time to think. I had been very busy for a few years and I was thinking about error handling and I was thinking about generic types. The other thing that you can find in the very first paper[11] on C with Classes is an explanation of why you need to parameterize things like vectors with types because you don't really just want a vector of doubles and a vector of int, you want a vector of T. That desire was in there very early and I explained merrily why you need it and how you could do it, claiming that macros would do the job. I got the right problem and the wrong solution because, again, macros is one of these solutions that works for one or two guys, working closely together and knowing what they're doing. It does not scale. If you have many users, relying on macros creates hell. So I needed something better. I knew I wanted it and I didn't know what it was. So I designed templates. Templates came with three fundamental design aims. I wanted them to be general. I had learned that I cannot imagine what people would want to do and I'd suffered from languages that were designed to make sure that you could only do what the designer thought proper. I didn't want to design one of those languages. So generality was very important. The second aim was performance. I would like to maintain what I had called "the zero-overhead principle", that is once you use an abstraction mechanism you'll run as fast as the best hand-coded alternative in the basic language. But, in particular, I really dislike the C array. The C array is the ideal abstraction of hardware's memory. You can't do better than that. It's a fixed-size sequence of objects. That's all it is. It doesn't know how many elements there are in that sequence. It converts to a pointer with the slightest excuse. And the pointers convert to other kinds of pointers with the slightest excuse and you get bugs. It's one of the major sources of bugs. So I'd like to put arrays out of business. I knew I would never be able to do that unless I could make an abstraction that could do a sequence of elements better than arrays but at the same cost as arrays. We're close to that with the current vector[12].

McJones: Right.

Stroustrup: I needed something like that. Those are two things: generality and low-cost abstraction, zero overhead abstraction. And finally, of course, I wanted interfaces; good interfaces so I can get modularity. Unfortunately, I had to give up on that. Nobody knew how to get all three and I thought I had no choice but to meet the two first criteria. The third criteria, decent interfaces, I (and I think everybody else) knew only how to do if you either had macros, which are unmanageable, or have something that's less general

[11] Bjarne Stroustrup: Classes: An Abstract Data Type Facility for the C Language. Bell Laboratories Computer Science Technical Report CSTR-84, April 1980 and *SIGPLAN Notices 17, 1 (January 1982), 42-51. DOI=10.1145/947886.947893*.

[12] The class template std::vector in the C++ standard library.

and have something that has a vector of functions, a jump table somewhere, which I couldn't afford if I wanted the performance of arrays for vector abstraction. That was a design problem that sort of defeated me at the time and we've been working on it ever since as you know. Last week they voted concepts out to national body vote and so we should have a technical specification of concepts that solves that problem later this year. It should have been in C++14. It will be in C++ in 2015. The C++ standards committee has never lost a national body vote. As a matter of fact, we've never had a negative vote in a national body vote. We work very hard for consensus. So we should get concepts and you can download it and use it today in a branch of GCC. Andrew Sutton built that one. He used to be my post doc; he's now a professor in [University of] Akron. This is a design that many people took part in (including you Paul). A lot of inspiration came from Alex Stepanov. Some of the stuff came from me in trying to simplify the way requirements are expressed in a language as opposed to what should be expressed and how you could express it, which mostly came from Alex.

McJones: That's really good news to hear. So this is a feature that's been 30-plus years in gestation.

Stroustrup: Yes, I blew it in 1981 or there about, knowing I wanted it but couldn't do it right. I came a lot of the way in 1988 when we got templates. Templates has been a huge success, but they have the weakness in how to specify interfaces which gives really bad error messages, complex codes, and other such problems that are unnecessary. We tried again in 2002 but got distracted by people who wanted something that was "truly object-oriented" or "just like Haskell". In both cases you get these jump tables, these vtables[13] in place and you need heroic efforts to optimize them away, and such heroic efforts don't tend to scale. That is, you can get really good performance of a POPL[14]-size program—one that fits in one column in a POPL paper. But if you are doing industrial-scale software the optimizations don't get good enough and you get costs going through an interface to an abstract class or something, which we can't afford, not if you want to compete with arrays.

McJones: I was thinking as you mentioned that principle of no-overhead performance that you could trace that back to the original FORTRAN project. John Backus wanted a language that scientists and engineers could express their problems in, but that the professional assembly language programmers at the time could not do better hand optimization, and he was actually able to achieve that by a balance of not overly complex language structures and a very good compiler.

Stroustrup: Yes. I mean this idea that things have to be affordable is really, really important. There are lots of areas where you can waste a lot of computing power. Our cell phones have 1000 times the performance on the PDP-11 that I started the C with Classes on and similarly with the little VAX machines that I completed the job on for the commercial release. It's a thousand times and more memory than that. People can afford to throw away 95 percent of that, or 99 percent of that, and [still] do a lot of things that

[13] Vtable refers to a virtual function table—see http://en.wikipedia.org/wiki/Virtual_method_table.
[14] The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages is usually referred to as POPL.

they want. So please don't get me wrong: I'm not saying everything should be efficient, but some things have to. FORTRAN lives because there are applications that take the biggest computers on the earth months to do and C++ survives for this kind of stuff because there are things like the video cards and the graphics, the networking software, the signal processing and such—all of this kind of stuff that's at the bottom of every system – has to be efficient. If you can run twice as fast in those areas, you need half the number of computers. If you look at Google or Facebook and their server farms, if you use a language that's half the raw performance you need twice as many server farms. This can be measured very easily. You just count one, two, three.

McJones: Twice the electricity. Twice the …

Stroustrup: … cooling, twice the maintenance, twice the monitoring, twice everything. I've talked to people who are concrete [specific, detailed] about what they can do on a machine and they really, really want the latest optimizations in C++ because they really need that extra factor of two or four or ten that you get compared to some other languages, or fifty for that matter.

McJones: Right. This has been very interesting, but we've jumped ahead of the story that you were telling us earlier. These features were some of the last things, the templates and exceptions that you had been thinking of, but around that time you were also starting to think that you needed to standardize the language, could you tell us a little bit about that?

Stroustrup: Sure. This is typical. We drifted away from the history and into the design issues.

McJones: That's fine.

Stroustrup: I'm just thinking back about what we were saying. No, I did not decide that we needed to standardize. A couple of guys—Dmitry Lenkov from HP and somebody from IBM, I don't recall the name right now—came to my office one day and said that I wanted to have ANSI standardization of C++. I tried to explain that this was a dumb idea, that the language wasn't finished, and I needed more time. They explained to me that you couldn't have a language in major industrial use where you could lose the whole design/implementation team because I got run over by a bus—a very nice polite explanation. But I still talked about how I need more time and they twisted my arm a bit more so in the end I went, "Oh yes, I'll do it, I'll do it, we'll do it next year," and we agreed on next year. The first meeting happened in December because—well that wasn't next year but it was close and anyway so that's how we started. I did not have the very correct idea that we needed standardization. I was too busy doing my own stuff. Now, ANSI and later ISO standardization is a huge pain in the neck with lots and lots of people involved, lots and lots of people working very hard, very often with no clear idea of where the language should be going in the future, very often with the idea that if we can slow down progress and change we can get something that's good and stable. Some would prefer to let somebody else do the innovative stuff; and others want

to turn C++ into the latest fashion. I mean, for all of the 1990s people wanted to turn C++ into a truly object-oriented language by jettisoning the C idea, except for the part of the committee that really couldn't care too much about the object-oriented stuff and would rather it was closer to C. So having direction in a standards committee is really hard. On the other hand, the guys [Dmitri Lenkov and friends] were fundamentally right: you need some organization that keeps the language [both] stable and evolving and that's very hard to do. I think things would have worked much better if we had had some resources so there could be a design group, an evolution group, that worked fulltime on where the language should be going, and then the committee at large would deal with the polishing and checking and compatibility issues, but that's not the way it is. There is a group in the committee that deals mostly with future stuff and there're others that deal more with everyday problems and there's a tension there. There's no resources for doing a serious experiment, for instance. We try and improve the process. We now have study groups that run in parallel and we have more internal groups, but now we have communication problems between the different groups and such. It's very, very hard. But you need a central organization. You need more than one guy or a couple of guys to run something this big, but organizing that is really difficult.

McJones: You've stayed through that process as the principal designer of the language, and yet other people have been contributing ideas and features. What does that feel like from your point of view? Was it sort of having your daughter grow up and get married?

Stroustrup: No, it's nice when people improve things. It's really painful when people try and take the language into a direction I really disagree with and then I have to try to stop them. It's really painful when I am certain about which way we should go and people say: "We can't do that. We should make it more elaborate and more complicated. We really must follow this fashion, not what you are saying." It's hard to get things through. Of course, sometimes they were right and I shouldn't go the direction I would have gone. I hope I would have discovered in time [by myself] and it would not have been set in concrete, but certainly the breaks [i.e., the committee acting as a break on changes] have worked at times. Sometimes we've gotten things [into C++] that probably shouldn't have happened because as an individual I can't be everywhere but it's very difficult. It's something that's under-estimated. I really understand why most language designers don't do this [standardization] or why [maybe] no [other] language designer do this. Other languages have committees that are disjoint from the original designer or designers. Other languages have "dictators for life" or something like that. Going through the standard process has its strengths and weaknesses and it's certainly very painful if you are a language designer, but possibly necessary.

McJones: From what you've said in the latest version of your language book, you seem to be pleased about where the language has gone.

Stroustrup: C++ 11 and C++ 14 is just a much better tool and one of the reasons actually is that I succeeded to some extent with my current push to make simple things simple. I was part of the people

who did the range-for loop like in many other languages. auto, which is like var or let in some other languages, is something I implemented in 1983-1984 and was forced to take out, again, partly for C compatibility reasons and partly because nobody could understand what it was good for. Similarly, we might get coroutines now [C++17], which, again, was something I was doing in the very early days[15] but haven't been able to get through. That's for the future, but some of the things people are most excited about C++ 11 actually is not the advanced features. It's some of the simplifying features so that they can write less code or repeat themselves less often and have fewer opportunities to make mistakes. I'm not a fan of the current trend of trying to absolutely minimize the amount of typing you have to do partly because then you have to know so much [about language details], there's no clue left in the code about what it is supposed to mean- [no redundancy to help catch mistakes], and partly because we read code much more than we write it. But certainly C++ has become too verbose. One of the things about concepts is that it actually simplifies the syntax of writing simple templates. So instead of writing:

```
template<typename I>
void sort(I i1, I i2);
```

you can write:

```
void sort(sortable& c);
```

This looks just like the function declarations we were used to in 1982 or thereabouts. Basically concepts takes the old view of a concept being the type of a type and generalizes it to say that there should be a set of types and values that should meet a set of criteria. You can say that merge takes three types that should be mergeable, meaning that there should be two input streams, an output stream, and the relationship between the input and output streams has to be just right. We can express that with simply saying mergeable{I1,I2,O} or something like that Again, I have this design idea that simple things should be simple and complicated things, well they'll have to be a little bit more complicated, and I hate making anything impossible.

McJones: Right. But in those examples with C++11 and 14 it's as if a scripting language style of programming has become possible. So it's a new paradigm in a sense.

Stroustrup: It could be. It's still strongly typed. As a matter of fact the code I write today is more strongly typed than it [my code] used to be. Furthermore, it's statically strongly typed, not scripting-style interpretation. One of the things we are doing with concepts is going away from the duck-typing model of

[15] Bjarne Stroustrup. A Set of C Classes for Co-routine Style Programming. Bell Laboratories Computer Science Technical Report CSTR-90, November 1980.

templates, which is what most of the scripting languages are using, into an interface-based model of computation which is what the rest of C++ uses and scripting languages by and large do not.

McJones: Yes, scripting is not the best word. Maybe, say, casual programming. It's this compactness that's very nice.

By the way, speaking of concepts you alluded to Alex Stepanov and STL earlier in the talk but it might be a good time for you to say a little bit about how you got to know him and what that has led to.

Stroustrup: Alex worked in some part of AT&T for a while and before that he had been at GE. He has moved around a lot. I remember discussing things with him pre-STL when he was at AT&T and he was trying to convince me to do things the way they're done in Ada because that was the language he was using at the time to express his ideas in.[16] I learned a lot from him, but I was not going to go that way. In particular, I was not going to have any statements for instantiating generic types. I was not going to put in the restrictions so that you couldn't have conversions. To this day Alex and I disagree on what you do with mixed-mode arithmetic. I really liked to be able to multiply a double with an integer. In a generic contexts that creates some complexity. But I also learned something about what could be done from Alex and the importance of being able to inline everything. Type information mustn't get lost. This completely plays into my zero-overhead bias in design.

Then Alex went away, and I implemented the first version of templates, and then something came back [from Alex] via Andrew Koenig, which was the STL. To be honest, I thought it looked really ugly and it certainly looked completely wrong. Like everybody else at the time, I had gotten the idea that things had to be object-oriented in the container world; that you had to have a hierarchy of containers and you had to have a hierarchy of iterators. I was working like that and of course STL is totally different. The STL is based on the notion of concepts. There just wasn't any language support for it [in 1988-vintage C++]. The STL is based on generic programming and it meets the zero-overhead principle because the code generated from it is really beautiful from a machine point of view. Furthermore, it was not just a container library, it was a container-and-algorithms library, or I guess from Alex's point of view it was an algorithms library that also had some containers. I looked at the STL for a couple of days, not being very enamored by it, and Andy was pushing the view that this really works. It turned out that I had over the years built up a set of criteria for what a good container library should look like. I actually had a list sitting somewhere with, I think, eleven points. You can find it in my third edition of *The C++ Programming Language*. Checking Alex's design against my list, I realized that apart from everything else about the STL, it met ten out of eleven of my criteria. I may get the number slightly wrong. Maybe it was twelve out of thirteen, something like that, but certainly there was only one thing left, and the one thing was having facilities common to all containers, things in common for all iterators. Given that, you could get debugging

[16] David R. Musser and Alexander A. Stepanov. *The Ada Generic Library Linear List Processing Packages*. Springer-Verlag, 1989.

[support] or serialization or something. Anything but that it [the STL] did, and it did very well. So I thought, “Well, yes, it looks odd, but it meets all the criteria, and nothing else does—actually all but one of the criteria. Nothing else does. I've been looking for years. I haven't found something like this. And then it does all this neat stuff with algorithms, and so, yes, this is good. The chances of the standards committee accepting it is of course very low, but it's so good it's worth trying.”

So we did a few things with the standards committee, including bringing Alex in. Alex is an absolutely first-rate rabble-rouser, and he got those guys excited in the way I've seen nobody else being able to do. And I kept plugging, Andy kept plugging, and after a couple of years we finally got it [the STL] in. That was very, very difficult. You had to convince a lot of people with different backgrounds. There were a couple of critical things, like Beman Dawes, who's one of the founders of Boost. In the middle of, I think the meeting after the one where Alex had explained things, when people were being somewhat rude about the STL, its complexity and its difficulty, and the ugliness, he [Beman] spoke up and said “You know, I thought that too, but since the last meeting I just thought I'll try and see what it was, and I implemented about a tenth of it, and you know you're wrong. It's not that complicated. It's relatively easy to do, and it’s cleaner than what we're doing.” That was very important. Again, the thing [to remember] about the standards committee: there’s lots of people. Sometimes they really help, sometimes they get in the way, but in this particular case Beman's contribution was essential. Another reason why it was very difficult to get a library like the STL in was that everybody by then had their own object-oriented container hierarchy. At one point Texas Instruments was going to propose their very nice object-oriented container library, and they were going to do it right after lunch. The rumor spread in the committee over lunch that they were going to do this, and after lunch I think we had five more organizations saying that if we are even going to discuss this proposal, we're going to discuss our corporate hierarchy also, and these “also” guys are people like GNU and IBM and Microsoft and Borland, so they are not lightweights. It was obvious that it [the standard container proposal] would run into impossible opposition. The fact that the STL was so different and the fact that it had the algorithms were probably crucial even to get people to look at it. It was a beautiful piece of work. It still is.

McJones: And so things fell into place with STL after these things you described. From that point on it became accepted, but the concepts issue was something that created a new demand for solving that problem, because now you have this rich set of algorithms that you'd like to describe.

Stroustrup: Yes. There's a huge pressure for concept that was building up from two sources. One was simply the bad error messages, and two from my perspective was that people got really enamored with the duck typing. Template provides compile-time duck typing, and so we get a lot of compile-time work and a lot of duck typing where we don't quite know what's going on and where the implementations of the templates sort of shines through because there's no interface and you have to know them. We have some horrendously complicated code that does amazing things, but people have trouble understanding it and [knowing] where the limits are. So my view is we need concepts for the traditional generic programming and for some of the template metaprogramming to introduce proper interfaces, and then we'll see what's

left after that. Expression templates[17] cannot be done well with concepts, because you are simply manipulating syntactic structures. They don't have semantic meaning till later, so that's a problem that's left.

The other thing that happens with templates was that people realized that templates couldn't just be used to generate new types. They could be used to generate values, and a lot of template metaprogramming is programming that generates values like 2 or 32 or 99 or something like that Some of the weirdest code is like that. Essentially all of that can now be done with what's known as constexpr functions, that is, functions that can be computed at compile time, which is a complementary approach that Gaby Dos Reis[18] and I did, realizing that there was a major need for compile-time computation both from the embedded systems programming and also [from] some of the stuff that we saw done with templates. Computing a value you just use a function. I mean, we know how to do that. That should be just that. People in the embedded systems programming world tended to use macros, and now you get to typeless programming, and people in the template world use templates, so now they get duck typing, and we really wanted ordinary functions where you give arguments of a given set of possible types and you generate results of the appropriate type and that's all. It should be like we're back learning our first function. And we're getting there. C++ is in C++11. It's heavily used. C++14 does it a little bit more generally, that is, you can now have a loop inside a constexpr function. That means the fundamental rule of a constexpr function is that it can be done at compile time. It means it cannot have any side effects, because the rest of the world hasn't been created yet, so obviously they must be side-effect-free. That means they're pure functions. It also means that any state that exists, exists only during the execution, and you get some really nice stuff out of that. So there are two approaches to limit template metaprogramming to where actually you want metaprogramming. The one is to take the generic code out with concepts, and the other one is to get the compile-time computation done with functions. Of course constexpr functions can be templates that can take concepts as argument types so it all fits together.

Similarly, the function objects that we have been using for years with templates and with the STL—I mean, function objects goes back to 1983, when I allowed the overloading of the application operator, and we have been using them ever since. There were versions of them that early. They've become essential with the STL, which is the efficient way of parameterizing with an operation, and then people come with lambdas, and you can define a function object right in the place. A function object in C++ is something with an application operator. Lambda is something that generates a function object in place and invokes it when necessary, so lambda expressions are a shorthand notation for the function objects that we'd had all the time, and they run as fast as function objects, possibly even faster if you do trickery with stack frames.

[17] See http://en.wikipedia.org/wiki/Expression_templates.
[18] Gabriel “Gaby” Dos Reis.

McJones: C++ now supports pretty rich functional programming and abstract data types, as well as object-oriented programming. All those languages that you studied in the 1970s are here and more in one framework.

Stroustrup: In one framework: that's important. That was why I emphasize that lambda is really a function object, and function objects have been there since before day one for C++, so it does fit together. There're a lot of barnacles there, which is complicating writing code, but there is an inherent logic there. And I was not deeply into functional programming in the 1980s, so this is relatively new. I knew the abstract data types, I knew the class hierarchies and I knew the close-to-the-machine programming. Functional programming is a set of ideas that has grown over the years, some of them coming out of the Lisp world, and now we have some of the aspects in C++, but it ties together with what's been there for a long time.

McJones: In your paper[19] of the third HOPL conference you mentioned three key properties of C++, and you sketched how you might be able to build a simpler language that would have the spirit of C++. Have you done any more thinking on that?

Stroustrup: Yes, I've done a lot of thinking about this. I think the key statement in that conclusion section of the HOPL III paper is [from memory]: “I think we can build a language with the expressive power of C++ and the ability to deal with hardware the way C++ does that's a tenth of the size, measured any way you like for size, and without loss of performance”, and I think that can be done. I have worked more on these ideas. If you look at the move semantics (coming mostly from Howard Hinnant, but following trends and dirty trickery that we used for years to get effects like that but they didn't), we can now build basically a closed system where we do not leak resources. So I talk about resource safety. People talk about memory safety. That [memory safety] is a relatively uninteresting thing. You can cripple a system much faster by leaking file handles than any other leak I can think of, including memory, so when I talk about resource safety, which is something I want, I'm thinking about file handles, thread handles, locks, sockets, the works, and I think we can do that now. Garbage collection solves one problem for the kind of world I have, and that is if you take a pointer to something is local and then store it and wait for the stack frame to go away, you have a major problem, and how to avoid that I'm not quite sure of. There are several ways, but I haven't decided which is simple, elegant, efficient, teachable enough, but, yes, I still think you can build a C++-like language that's much smaller than what we're dealing with now. I would not throw away C, but I might throw away some of the syntax of C. C is a model of how to deal with things. It has objects, it has pointers, and that's very important. It maps to the memory, and you need that for a C++-like language.

[19] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). ACM, New York, NY, USA, 4-1-4-59. DOI=10.1145/1238844.1238848

McJones: We left your personal career, and we focused more on the language. Maybe we could loop back. A dozen or so years ago AT&T ceased to be the right place to be, and you made an interesting career transition.

Stroustrup: Yes. AT&T Bell Labs at the height of its powers in I think the 1980s and 1990s was just a unique place in the world. There was no place like it. In the Computer Science Research Center we had between 30 and 60 people during that period, and many of them are still my friends. An extraordinary number became famous in one way or other, IEEE fellows, ACM fellows, members of the National Academy, and things like that. We were just doing all sorts of interesting work at the time. The environment made that possible, Bell Labs, the philosophy that you could let smart people work on things they thought interesting rather than having a command economy. We had a management structure that kept that from going out of control, it put a lot of emphasis on really good department heads, really good directors, so that you actually get something out of a system like this. The idea that you have a mixed portfolio of research so that something comes along every year or so, but that doesn't mean that you can only work on things that'll come to fruition next year. A lot of us were working on things that take five or 10 years to come out, but every year every department delivered something of interest, and managing that was difficult. The environment was really nice. A lot of the people were nice people. A lot of the people are still my friends. Of course there were people with enormous egos. There's no shortage of enormous egos in the computer industry, but many, many people had no need to have a huge ego or be rude, and they were some of the nicest people I've met. That was a good time. I brought up my children near Murray Hill in New Jersey. People have the strangest views of New Jersey. If you try and land there in a plane you can't find it for the trees. That is not the image that people usually have, but it's a good place to bring up children. It was a good place to work.

Then “Wall Street” got an idea that vertically integrated companies couldn't possibly work, so you need to chop them all to pieces. They also had the idea that everything that can be invented has been invented, so you don't need research or advance development organizations, so you have to cut that. Both theories are wrong, but it hurt Bell Labs enormously, and it became a far less interesting place to work. First the Bell system was broken up, and then AT&T and Lucent split ways, the idea being that some people built the switches and the other people built the software for using the switches How it makes sense to separate it [design and use of switches] there, I don't understand, especially since my background is working in exactly that [hardware/software] interface. Anyway, then we had some good years at AT&T Labs. It was in Florham Park, just on the other side of the Great Swamp from Murray Hill. Then we got hit by another set of cutbacks forced by Wall Street that didn't have much foresight about technology, no tolerance to delays from research, and was misguided by false information given by WorldCom. They [WorldCom] got fined later for giving bad information, but it hurt the labs badly. We lost 40 percent of our PhDs in one massacre. I was a department head at the time, and they laid off two of my people without telling me, possibly because they knew that I would've hit the ceiling very fast if they had told me. After that I was open to new challenges. I wasn't laid off. I had a nice corner office and all that, but the oxygen had been sucked out of the air. I noticed that the people left behind were on average less interesting than the people that left. The people that left were the troublemakers, the people that would pin down an

executive in the corridor and tell him he was all wrong. Those guys left, and those were the interesting people to talk to.

It so happened that our youngest son had just gone to university and was settled in there. So the kids have gone to university and had a lot of fun, so why shouldn't I? So when some of my friends suggested I should go and become a professor I went “Sure. I'll try that.” Now I've been in industry for 24 years and I actually have something to teach the students, as opposed to when I just came out of school. It happened that they [my friends] were working on some interesting problems at Texas A&M University, and this was a department that really wanted to improve. We [The TAMU Computer Science and Engineering department] have gone some 20 spots up in the rankings since I joined, so we did do something good. We rejigged the whole curriculum, and I tried to do academic research, I tried to teach freshmen, I tried to teach graduates, I tried to supervise PhDs, all of this good stuff. This was interesting. But after 10, 11 years it was beginning to go a bit stale, and also I have some problems with academic research. It has a tendency of drifting away from real problems. You abstract from a real problem, then you abstract from the abstraction, then you solve the abstraction of the abstraction of the abstraction, which may or may not relate to the original problem. I had some problems there. Some of the stuff I thought was really interesting to work on was unpublishable. I once wrote a paper on how to do abstract syntax trees 30 times better than GCC, and the paper was rejected for two reasons. The first was that we had misunderstood C++. This was me and Gaby Dos Reis, who was the French representative on the standards committee and one of the best people on the standard. Also, he was the shipping manager for GCC. They claimed it had been done already in GCC. I mean, this was totally bogus. When we complained they came back and said, “Oh, that's [just] engineering; we're computer scientists.” If we had been in engineering for cars and come up with a 32-times improvement of one of the key components of an important gadget, we'd have been heroes. Here, it took several more attempts to get the stuff published, and, by the way, it's now in use in various compilers, but these kinds of things were grating. These things are discouraging, and in particular it makes it hard to bring up students well, because if you want to graduate as a PhD you need to have publications, and de facto that means you have to follow fashion. And, well, following fashion is not that fun.

So I was beginning to think it’s time to get back to industry, back to the kind of problems that I know and like and get them in the real form rather than the distilled form and cleaned-up form you get in academia. Again I was open to suggestions, and it so happens that every year since the 1980s I've been going around giving talks in industry and academia. Basically, I exchange a talk about what I'm doing for an explanation of what their problems are. That's one way of making sure that when you work on a design, when you work on evolution of something, it actually has a direction. In 2013, I went and gave a talk at Morgan Stanley in New York City. After my talk the not unusual suggestions started coming that maybe I would like to go and work there, and my answer was “Of course not. I mean, you're a bank. The kind of stuff I'm doing is not what interests you.” [To my surprise, they responded:] “Yeah. Oh, yeah, yeah. This is fine. We like your kind of stuff,” and they pointed out some of the things I'm doing that they're interested in. So I said, “Oh, my time horizon is all wrong for a bank. I mean, some of the things I really would like to do take at least five years.” [They answered:] “Oh, no problem.” So we went through the song and dance,

and eventually they wore down my usual defenses. They had one secret weapon that they didn't even know they had. Of all the organizations that had the properties that I could work for, they were the only one who lived within 20 minutes of my grandchildren, so I'm semi-logical about this kind of stuff. I have a set of criteria. They met them. Other people meet those criteria, but they [Morgan Stanley] were in the right place. So I'm now sitting in my apartment, which is halfway between my grandchildren and my work.

McJones: That's a pretty nice place to be.

Stroustrup: Yes.

McJones: So at Morgan Stanley you are designing future versions of C++, thinking about other programming languages, and what else?

Stroustrup: I spend a lot of time looking at problems in the bank, a lot of them to do with networking, a lot of them to do with how to express interesting coding problems. A lot of them have to do with performance. A lot of what I'm doing is fairly close to the hardware but not everything, so in some sense from that perspective it's the same as ever. They have different kinds of reliability problems. I'm always interested in reliability, in dependability, and they do have the interesting problem that you can't lose a penny. The answer has to be correct to the last penny. If you're doing a nice video system, if you lose a frame it doesn't matter too much. Even if you're doing a fuel injection you can't lose a cycle, but if you're ever so slightly off for a single squirt, the engine will still keep running, and you can correct it for the next cycle. With money you can't do that. It's against the law. You have to get it right every time, not just on average, so there are interesting constraints on the problem, but basically performance, reliability, correctness, same kind of stuff [as ever], and so I'm now closer to reality, and I can do better language design. I'm still part of the standards committee, and I'm also a professor at Columbia now, so tomorrow I'll go and give my lecture. [Curiously,] leaving academia meant that I'm now actually a professor twice over, because they wouldn't let me leave Texas A&M, but most of my work is in the bank and working with the code bases that we have there, and that informs my work on the evolution of C++ and my thoughts about what can be done better.

McJones: You mentioned the challenges of being in academia, but do you have some advice for a future grad student who would like to impact the world of programming language design?

Stroustrup: Oh dear. There are a lot of people who come with the firm idea they want to build a new language or they want to build a new game or they want to build a new operating system, and what is lacking in most cases is an understanding of what's the problem. That is, C++ succeeded not so much because I made a beautiful design but because I was really lucky of having identified a problem that a lot of people were about to get, namely handling complexity with larger memories and faster processors. We were then starting to tackle more complex problems, and I was trying to deal with complexity, but

complexity under the constraint of performance, which was what people didn't get. So that's the fundamental problem that I decided to attack, and I was lucky I wasn't alone and my design was good enough for a lot of people.

So when people want to look for a problem I think first of all they shouldn't look for a project to change the world right away. They should try and understand the set of problems and then come up with solutions. You can do very particular solutions trying to meet the deadline for the next quarter, and that's probably the best way of getting promoted, and soon you'll become a manager and you'll never do something really useful again. Or you can start learning from the experience, generalizing the problem and seeing “Is there a solution that can help a broader spectrum or a larger group of people, larger set of problems, larger time scale?” So focus on the problem and use the initial projects and your knowledge from university to come up with a solution there. I think all of our theory and all of our courses help us solve the problem, but we have to find the problem first, one that's worth solving and find the constraints on the solution to the problem. After that, we have pretty good ways of engineering a solution.

McJones: You orchestrated the donation of the Cfront source code to the Computer History Museum. We're really grateful for that. What are some of your ideas for how the museum can help researchers, students and educators?

Stroustrup: First of all, I should point out that getting Cfront into the archives was your idea, and I thought it was a splendid idea and thank you. I just collected some of the pictures and the source code and the manuals and such, so this was good.[20] I think that students are by and large ahistorical. They have not been taught history, they don't understand history, and they think that history has nothing to teach them. Everything was invented yesterday, and everything is the latest fad. I think that that is most unfortunate, and I think the schools are encouraging this. There's a lot of schools that, for students wanting to be engineers, look only at their science and engineering scores, forcing them to not take, say, history in high school, not to take a liberal arts education in university but just do the thing that's required. Also, there is a lot of emphasis in the press and elsewhere about the best career being the one that gives you the greatest amount of money soonest. That is not a long-term motivator. The people that come through this successfully are often not very nice people. They have spent all of their childhood, youth, and early adulthood in a very narrow area and focused on the career to the exclusion of everything else. That's not good. I think people should know more general history and they should be better at communicating. Probably the best way of getting to be better at communicating is to read a lot and to listen a lot to other people.

I think the Computer History Museum can help motivate by actually showing some of the things that were done in the past, and especially if they focused a bit more about why things were done. I went through the Computer History Museum some time ago. I went through some of the rooms: “here is Model X of

[20] See http://www.softwarepreservation.org/projects/c_plus_plus/.

computer from Factory Y, here's Model Z of a computer from there", and it looks like one washing machine after another, and that very soon gets stale. Yes, I was excited. I realized "I programmed this one. Ooh, that was a fun one. Oh, I couldn't get that one to work." So if you can go through like that, there is something there, but for most people there's not something that motivates them. It doesn't tell why that thing was being built, what are the kind of problems it was used for. Showing just hardware, I think is uninteresting, and there's a tendency of focusing on hardware because you can see it, you can knock on it, it's pretty pictures, but a computer without software is kind of a boat anchor, right? And so more emphasis on software, not just languages but "What did that computer do for a living? How did people interact with that computer? What kind of problems was it used to solve? What kind of tools did people use in addition to the hardware to provide those solutions?" I think a greater emphasis on that would be helpful. I may simply be behind the times, and the Computer History Museum may be looking far more like what I'm describing than it did when I went through a couple of years ago, but I don't want to see another dishwasher-like piece of hardware out of context. I think as a museum you have to collect artifacts. Compilers, databases, application programs should be part of those artifacts. Since there are many, many, many pieces of software for each piece of hardware there should be more software in the museum than hardware if you want to reflect reality. But once you present it you have a challenge, because software you can't see, you can't feel, you can't knock on it, and so I think the context of the computer, the use of the computer, the problem that was to be solved by the computer is probably one way of addressing that.

McJones: The Revolution exhibit is certainly better at giving context than the original Visible Storage Area. I'm not sure which one you saw. They're working on a major software exhibit that's trying to show maybe a dozen applications and systems in context. When that's available hopefully you'll get a chance to come out and see it.

Stroustrup: Sure.

McJones: I just need to correct the record. It was really Alex Stepanov who inspired me to collect not only C++ but a lot of the other source code too.

Stroustrup: Alex is at the root of many interesting things. I didn't know he had prodded you in that direction.

McJones: Yes, that he did.

Stroustrup: I certainly know it was you who prodded me so that I delivered the stuff. Thanks.

McJones: And I greatly appreciate it. I think that wraps up most of the things that I was hoping that we could talk about today.

Jon Plutte: I'm going to ask one question, which you already covered. It's fascinating that David Wheeler and Maurice Wilkes and you are all Fellows. Think about speaking to the audience at the Fellows Awards and how cool you think it is that you are part of a family line at this point.

Stroustrup: I mentioned earlier history and the importance of history, and I don't think history is something dead. It's part of our world, it's part of the way we understand the world, and so for that reason I have known about the Computer History Museum and its Fellows for many years. It's one of the things I've been looking at. I've been there physically, I've been there on the Web, and I find it's really cool. I mean, not only being a Fellow there, but having been part of all of this and being recognized to be part of all of this is very, very exciting for me, and I'm very honored. I'm probably the first third-generation Fellow, that is, my advisor was a Fellow and his advisor was a Fellow. We're still a young field, so hopefully there'll be many more third-generations eventually, but that struck me as interesting. I actually went and read a fair number of the short stories that go with the Fellows. In many cases I was sort of wondering what else is there. I'm very slow to watch movies, because they move fairly slowly, and I'm busy or I'm tired or something, so if there had been some transcripts of their presentations that was linked from the bios that would have been really useful for me. Not everybody can do what I do: For some of them [The CHM Fellows] I can sort of remember them [and/or their work]; they're real, and the biography is something that prods my memory, and that's nice. For others a little bit more information, the videos of things like this will help. Transcripts, if available, will help people that are busy.

Yes. And of course if I didn't say it already I am very honored, and this is very important to me. Thank you.

END OF INTERVIEW

