Interview by Julia Schmidt from “Heisse Developter” with Bjarne Stroustrup. August 2014.

- How did you get into programming?
    - Looking for something to study in university, I aimed for some form of applied mathematics and chose computer science. Fortunately for me, Computer Science wasn’t applied math, and I quickly learned to love programming, machine architecture, and operating systems. It just was – and is – an amazingly dynamic and expanding field. The fact that it is a field in which an individual can make a positive contribution to the world is important to me.

- Looking back at over thirty years of C++, what are the things you are most proud of when it comes to what it has made possible? What are the things that make you throw your hands up in horror, when looking back at the whole journey?
    - I think the best thing I did for C++ itself was simply constructors and destructors. They were part of the design in the first week. Most modern C++ techniques depend on them. I’m most pleased with seeing C++ having a role in many of the great scientific and engineering breakthroughs of our time, such as the sequencing of the human genome, the Mars Rovers, and finding Higgs’ boson. Having contributed – ever so slightly – to such historical endeavors is part of what keeps me going. It is also exciting – and a bit scary – to know that C++ massively used in just about every industry, such as communications, transport, agriculture, and of course computing; it is a critical part of the world’s infrastructure and touches our daily lives. I write this using software written in C++, on a computer designed and implemented using C++, and will transmit the final text using communication systems implemented using C++. Obviously, not all code we use is C++, but most often some is. Finally, C++ has provided a stimulus to programming language an programming tools development, partly through its direct contribution and partly through spurring people on to do better.

      Obviously, C++ has also been used for applications that I do not like, but let’s not go there. Any powerful tool can be used for both good and evil. I have seen more horrible C++ code than I could possibly have imagined! Horrible code is of course not just a C++ problem, and maybe the worst code I have seen wasn’t C++, but I feel a bit of responsibility for the C++ code. I suspect that problem is far bigger than C++; it is a failure of establishing Computer Science as part of a profession of software development and a failure of education. It seems that our education systems produce some scientists, hordes of semi-skilled “coders,” and far too few people with a balanced set of software development skills (e.g., a grounding in mathematics, data structures, algorithms, machine architecture, design and programming techniques, systems development, testing, quality control, and a good grasp of an application area) and a professional attitude. Trying to compensate for that through programming language design is possibly impossible and – worse – makes introducing batter techniques and upgrading our tool chains (including our programming languages) harder.

- In recent years, languages like Go or Rust have been positioned as C++ rivals. Why do you think C++ still has a place in modern day programming? Are there things it can learn from its younger competitors?
    - C++ can learn from younger, as well as older languages. We try to learn, and sometimes make progress. Conversely, newer languages learn from C++.

      It is hard to compete with C++ in its code domains (e.g., see the Google language comparison http://www.computing.co.uk/ctg/news/2076322/-winner-google-language-tests) and C++ is not standing still. We have C++11 and now C++14 was delivered on time. C++ Is no longer the 1980s caricature that some people seem to love to hate.

- Where do you see C++ in fifteen years, when it has been around for roughly 50 years?
    - It is not unlikely that in 15 years it will still dominate areas of programming that require serious attention to resource consumption and dependability. If not, some language that has learned a lot from C++ will.

      By then, I hope C++ will offer a rich selection of simple and specialized concurrency models. I hope it will have a cleaner syntax, be more type safe, and compile faster. I think is will still have a direct and efficient map to hardware, very general zero-overhead abstraction facilities, and be even more precisely specified. I expect the infrastructure of supporting tools, libraries, educational facilities, and community support will have improved immensely.

- What are the main reasons developers should start to use C++14, seeing that many compiler companies still haven't completed their C++11 support?
    - Actually GCC and Clang claim 100% C++14 compliance today and Microsoft C++ isn't all that far behind. All the implementers treat C++11 and C++14 as one. That is the right approach, C++14 is an incremental refinement of C++11 – deliberately designed to "complete C++11", rather than breaking significant new ground.

      The adoption of C++11 and C++14 has been much faster than the adoption of C++98 were. The compilers are much better sooner now, there seems to be better communication about the new features in the community, and many of the new features simplify programming without requiring massive overhaul of a code base.

- Do you think it is a good sign for the next iteration of C++, that C++14 was done so quickly? Or was this only due to the relatively few changes when compared to the last version of the language?
    - When you ship a major product – such as the C++11 standard – there are features that can't be included because of lack of time. Some implications of a design you learn only through initial use of the product. C++14 was define to be a "minor release" to deal with such issues. However, C++14 is not just a "bug fix release."

However, its scope was deliberately restricted so that we could ship on time. For example:

    - C++11 gave us lambdas; C++14 allows generic lambdas
    - C++ gave us user-defined literals; C++14's standard library allows "**Great!**"**s** as a **std::string** literal (to contrast: "**old**" is a C-style string, a zero-terminated array of characters) and **2s+45ms** as a time **duration**.
    - C++11 gave us **unique_ptr**; C++14 added **make_unique()** to create an object and return a **unique_ptr** to it.
    - C++11 gave us **constexpr** functions to simplify and generalize compile-time evaluation; C++ allows **if**-statement and loops in **constexpr** functions to make them far simpler to write.

In all, there are one or two dozen additions (depending on how you count). In addition, the standard has been significantly clarified for implementers, so portability is improved.

- Do movements like Continuous Delivery influence the way C++ is developed?
    - Yes. Some of us (Notably, Herb Sutter from Microsoft, the committee's convener/chairman) decided that the "one new standard decade" model used by most (all?) ISO standardized languages was not sufficiently flexible and responsive. So we chose a model based on the idea of a major release followed by a minor release and set a very ambitious goal of three years for the first minor release (C++14). We are aiming for a major release in another three years (C++17) and we just might make it.

      In addition to the ISO standard proper, the committee works on Technical Specifications (TSs). We see those as an intermediate step to the standard. They are implemented and made available so that people can use them in real code before final standardization. That was what was done for types such as **unordered_map** and **shared_ptr** before the C++11 standard. There are about 11 TSs in various stages of completion. For example:

        - Concurrency
        - File system (complete)
        - Concepts
        - Parallelism
        - Networking
        - Transactional Memory
        - Ranges

      See https://isocpp.org/std/the-committee for a complete list and the current status).

      Concurrency and Parallelism will – I think – be the most important additions to C++17, or possibly modules, but I expect that the most significant language change will be "concepts." Concepts are specifications of requirements for template arguments. They will change the way we think about generic programming and library design. Concepts are already shipping as a GCC branch and more implementations are in the works

- What's your opinion on Apple's Swift?
  - I don't know enough to have an opinion.

- Are there any other new languages, such as Julia or Elixir, that you find interesting in certain ways?
  - There are lots of interesting ideas in the language design world, but I get my major inspiration from system building. I came from the "systems" area of computer science and I feel more comfortable talking about distributed systems or cache effects than about type theory. For me, the applications are far more interesting and important than the language features. By itself, an individual programming language feature is boring.

- As there are some programmers out there, who claim C++ is hard to learn, what do you think could be done to make it easier for people who are new to the language?
  - Compared to many languages, C++ ***is*** hard to learn well. There is more to learn because C++ can express more and is used for a more diverse set of application areas than most languages. That learning can be most rewarding, though, because it allows for an unsurpassed combination of flexibility and performance. C++ exposes the fundamental mechanism that other languages tend to hide – leaving their programmers believing in "magic."

    However, this is not about language design, but about learning. C++11 and C++14 are far easier to teach and learn well than C++98, and C++98 is far easier to teach well than it is reputed to be. The problem is than many novices head straight into the darkest corners of the language, ignoring simple solutions. Similarly, many teachers insists on exposing innocent students to every obscure detail of C before showing the C++ facilities that allow people to ignore those complexities until they are needed (if ever). C++ facilities, such as **vector** and **string**, simplify much code that is often written using arrays, pointers, macros, and explicit memory management.

    When I teach C++ to beginners, I use the modern C++ facilities from day #1 and don't expose students to arrays and pointers until eight weeks later. See my *Programming: Principles and Practice using C++*. It has been used for many thousands of students in classes as well as many thousands who study programming on their own. There is a German version. The current English edition is the second edition using C++11 and C++14 to simplify the introduction of programming concepts and techniques. The German 2nd edition is in the works.

    The other source of difficulty in learning C++ is the compatibility with C and with earlier versions of C++. We can't just ignore the facilities that were introduced in the 1900s. There are billions of lines of C++ "out there" and most don't conform to the most modern taste. Stability over decades (compatibility) is a major feature, and ISO C++ offers it. However, we should not emphasize the oldest features when teaching beginners – but many people do. Unsurprisingly, then things get difficult fast.

We can do much better. C++11 facilities such as range-for loops, uniform initialization, lambda expressions, generalized constant expressions, and of course the larger standard library can lead to much simpler code. Such code is also easier to read, write, and maintain. I can’t describe these new features here, but there are many examples on the web and my recent book *A Tour of C++* gives an overview of all of C++ (language and standard library) in just 180 pages. If you are an experienced programmer, you can read it on a long plane or train trip. If you are a real novice, you probably should look at PPP first.