Hello!

- How did you learn to program? When did it all start?
  - In university. I signed up for “Mathematics with Computer Science” without a clue about what that was (The Danish name for that course of study did not include the work “Computer” and I was under the impression that it was some sort of applied math). Once I encountered computer science and in particular programming, I never looked back. It was just so amazing what you could learn and what you could do. What I learned about machine architecture, data structures, operating systems, and compilers was put to good use in later years.

- What was the first serious program that you wrote?
  - Starting in my third year in university I had a series of small commercial programming jobs that financed me through to my master’s degree. The systems were small and tailored for specific small businesses. They taught me the process of design and implementation from requirements analysis (with customers) to deployment and support. I saw the disasters that could happen to individuals if I didn’t do my work well – that was what made these programs serious.

- Can you describe the program that you wrote during your lifetime, that makes you the most proud of?
  - Cfront, the first C++ compiler. It did lexical and syntax analysis building an AST, then it walked that tree to do type checking, and again to do a few source-level optimizations, and finally to generate C for portable code generation. It was designed for a megabyte of memory with a 1MHz CPU. It turned out that it could manage with even less: I ported it to the original PC. It was a very portable program. One reason it could manage with so little memory was that it didn’t store anything for longer than it needed the information and that it never leaked.

- On what kind of software development projects are you working currently?
  - For Morgan Stanley, I am mostly concerned with reliability and performance of rather large distributed systems. For the ISO C++ standards committee, I work on a variety of proposals for C++17 and beyond. For example, I am working on proposals for “smart references” (based on an operator.()), automatic generations of comparison operators (such as == and <), and a uniform function call syntax (to ease interoperation of object-oriented and generic code). You can find them by typing WG21 into your browser and looking for papers. Alternatively you can look for my publications page (on www.stroustrup.com) or the C++ Foundation’s web page: www.isocpp.org.

- What programming languages and tools are you using at your work nowadays?
  - I primarily use C++, but the systems I work with use a variety of languages.
• Frederick P. Brooks once called programming a "tar pit"... What is your opinion regarding current condition of Software Engineering as a craft? What are the strongest threats and opportunities in that filed?
  o Programming can be a “tar pit” (the “Turing tar pit” in Brooks’ words) if you make a mess of it. We really have to do a better job at constructing proper abstractions for our systems and to implement those better and more directly in our code. C++ has (and always did have) two complementary parts: a close-to-the-machine part (originally borrowed from C) and a set of abstraction mechanisms (originally borrowed from Simula) to allow us to get to a higher level of programming. Both the ability to map to the machine and the ability to abstract from it are vastly improved in modern C++ (e.g., C++14), but the fundamental idea has remained constant over the years.

  It is really a waste to write every part of a program in terms of bits, bytes, and pointers hoping for performance and reliability. We can do much better with a more structured and systematic approach using C++. Sometimes, we need to manipulate bits and machine addressed (and C++ is exceptionally good at that), but we should not do that all the time – or we end up in an unmanageable mess of pointer spaghetti and error-prone data structures.

• Can you try to foresee the Software Engineering and programming language evolution in the forthcoming years 10-20 years? What do you think about the functional programming trend?
  o “It is hard to make predictions, especially about the future”. No, I'll not try to make any firm predictions. There are currently people who believe that functional programming is the cure for all problems, just as there were people who believed that object-oriented programming was the cure for all problems. My view is that there is a lot to learn from FP just as there was from OOP, but that the fanatics are wrong as usual. The most successful “FP languages” are not 100% functional.

  C++ had non-OO features from day #1, and boy did I get criticized for that! Generic programming, as represented by the STL, has a lot of functional aspects: Function objects have been used in C++ for decades where functional languages use lambda functions and closures. C++11 provides lambdas as a convenient notation for generating function objects. Constexpr functions are pure. From 1984, C++ provided const for representing immutable data.

  The FP community owned large sectors of the education establishment for 30 years without noticeable impact on computing practice. I think the proper policy is – as ever – to carefully adopt what has been demonstrate to work and translate it into something that can be applied on a large scale in industrial settings. To quote Kristen Nygaard (the inventor of Object-Oriented Programming): “If a PhD from MIT is required to use our language, we have failed.” (from memory)

• How about your writings? Do you work on some new book or books? What about *C++ Programming Language 5th Edition* covering C++17?
  o I maintain my books: The C++ Programming language got a completely rewritten 4th edition last year, Programming: Principles and Practice using C++ got a 2nd
edition this spring and now we also have the new *A Tour of C++* describing all of ISO C++ and its standard library in just 180 pages (not in the greatest of detail of course). That’s enough for just now. I will reconsider once C++ has evolved sufficiently to warrant newer editions. Writing such books is an incredible amount of work.

- **What can we expect to see within incoming C++17 standard?**
  - Mostly better high-level mechanisms for concurrency and parallelism – that’s where hardware drives us. Several new standard libraries (e.g., networking and ranges). Concepts and maybe a concept-based STL. Maybe modules (faster compilation). Maybe fast, simple co-routines. Almost certainly many small extensions. I hope the “small extensions” will lead to simpler programs the way the “small extensions” in C++11 did: auto, range-for, lambdas, initializer lists, etc.

- **How do you see the target for C++20?**
  - A committee of hundreds of volunteers have a hard time setting technical targets.

- **Does C++ Standard Committee has some kind of longer term strategy for C++ standard development? Let’s say in 10-20 years perspective? Can you foresee how C++30 would look like?**
  - No. I try to set a direction for C++, but it is not easy. Compatibility/stability is a feature. We cannot afford to take significant compatibility breaks or long delays waiting for perfection. We need an evolutionary strategy where the language is a viable choice for millions of programmers at every point along the way.

  I hope that the surface complexity of the language will decrease. I have reasons for that hope: C++11 and C++14 programs can be much simpler, shorter, and faster than their C++98 alternatives. Backward compatibility is essential, but writing code in antique and suboptimal styles is not.

  We can do without exposing raw pointers, new and delete (think: scoped objects, make_unique, and make_shared). We can eliminate resource leaks (using RAII and move semantics). We should eliminate macros and most casts: that would finally allow tools developers to dramatically improve C++ tool and IDEs. We should have many, many more high-quality libraries.

- **Do you see the future of Rust as an C++ alternative? May it be a kind of “threat” for C++ position in the future?**
  - No. I like the idea of Rust trying to be a safer language on the C++ model of resources, but I don’t see it escaping from its niche status. There are so many good languages “out there,” they adapt to meet new “challenges,” they adopt new ideas, and it is far harder to make people use a language for significant projects than it is to design and implement it.

  It is good to see new languages. We need lot of ideas because the state of the art in systems building really isn’t that impressive.

- **How does the standardization committee work? Is it only about meetings?**
The committee works continuously. There is hardly a day where I don’t see messages on the internal reflectors where discussions take place. People work on proposals between meetings. There are three major (week-long, intense) meetings a year and smaller more-focused meetings in-between (face-to-face and telecom). In all, I guess 250+ people take part. They are all volunteer. If you want to try to keep up, start by following www.isocpp.org.

We are currently working with two organizational innovations: the train model of shipping at frequent fixed times and the idea of first formally specifying facilities as ISO “Technical Specifications” (TSs) and then transferring the facilities into the standard if/when they have been seen to prove themselves. Shipping what is ready at pre-determined times was suggested by Herb Sutter as a response to the problem of the conventional 10-year gap between ISO language standard releases. It seems to work. We shipped C++14 on schedule only three years after C++11. C++14 is a minor release, but still offers significant improvements over C++11 (it completes C++11). Significantly, the major implementations (e.g., GCC, Clang, and Microsoft) already ship all or almost all of the C++14 features. The first post-C++11 TSs are approved and more are close to ready.

- How do you feel about your fluency in C++? Do you have any gaps or weaknesses in your knowledge?
  - My fluency with C++ is fine. I don’t try to be a language lawyer all the time and frankly some of the more clever uses of C++ horrify me: I’d hate to maintain them or have my life depend on their correctness or performance. Try not to be unnecessarily clever when expressing your ideas!

  I have lots of weaknesses and lots of gaps in my knowledge, of course, but they are not in the area of mainstream C++ programming.

- What do you think about what you have done already from the time perspective?
  - I guess that’s not for me to say, but when I started, function declarations with argument types were yet in C’s future, as were const. OOP was unknown or considered too difficult, too slow, and too specialized for real world use by most people in industry (and academia). Generic Programming was unknown or considered a pipe dream (and part of functional programming). It is hard for people today to imagine how primitive the state of the art in real-world programming language use were and how distant and unknown the research systems we admire today were from major use. I think I did my bit to make the change to today. When we consider major changes to the way people think, we have to think in terms of decades.

- In 2003 there was a first draft regarding concepts… and still nothing has been really done about them. Also, they were rejected as a part of C++11. Are there plans to introduce concepts sometime in future?
  - Concepts will be an ISO C++ Technical Specification in 2015. It is not the failed 2003 design, but a newer, much simpler, design based on the idea of concepts as predicates. An implementation exists as a branch of GCC, a specification exists as a mature ISO draft, and other implementation are starting up (Clang and Microsoft).
The current design of concepts (sometimes referred to as “Concepts lite”) was done by Gabriel Dos Reis, Andrew Sutton, and me (working together at Texas A&M University). The GCC implementation was done by Andrew Sutton. The design was based on earlier ideas from Gabriel Dos Reis and me and a lot of work and encouragement from Alex Stepanov (the father of the STL) and many others. For more information about the history of concepts, see my publications page and my page for WG21 (ISO C++ standard) contributions.

- Memory safety is one of the most popular issues in software. Do you think if current memory safety solutions existing in C++ are satisfactory? Do you plan to do something in order to improve that in future?
  - We don't just need memory safety, we need general resource safety. Not leaking memory is an important, but incomplete, solution: leak a thread handle, leave an unused key in a dictionary, or leak a socket and eventually the system will slow down because of starvation and eventually stop. Non-memory resources must not be leaked and they must not be retained for much longer than necessary. C++'s resource model (scoped objects, RAII, move semantics, smart pointers) provides that. A garbage collector by itself does not. Thus, garbage collection by itself is neither necessary nor sufficient.

Eliminating resource leakage is necessary, but there are other ways to corrupt objects: for example unchecked unions, casts, and lack of range checking. I believe unsafe unions and casts can be eliminated. I rarely use those these days, we can build efficient tagged (discriminating) unions and templates can reduce the use to casts to hardware interfaces. Commodity hardware will soon offer cheap range checking for the cases where the compiler cannot guaranteed the absence of range errors (as it can for a range-for loop, for example).

I think we will see guaranteed resource-safe and memory-safe C++ in the future, but I'm not willing to put a date on that.

- C++ is getting more and more complicated. Restrictions are being constantly reduced, but for the price of the language complexity. Do you see that as an issue?
  - The total language becomes more complicated; that's a necessity as long as stability is a feature. However, what really matter is not language complexity, but the simplicity of applications. With better libraries (enabled by better language facilities) and better/simpler/safer language features we can simplify application code. So, it's an issue, but a manageable issue that is being addressed. Just compare C++11 code to C++98 code – and try to concentrate on simplicity, rather than cleverness. Measure before you make performance claims.

When teaching and learning C++ we must break with the bad old habit of trying to understand every little detail at once (that is impossible for every modern language anyway) and in particular we must not try to learn C++ in a bottom up (“all of C first”) manner. Doing so is most wasteful and often lead to poor programmers.
• What are the biggest problems of C++ in current shape, how do you see the perspective to solve them?
  o The biggest problem with C++ – by far – is ancient code and ancient programming styles. There are still people who are convinced that “pure object-oriented code, relying exclusively on class hierarchies” is the sole ideal for C++ programs! Similarly, there are still many people who believe that efficient code must be low-level.

  We need to convince people that it is in their interest to move to newer styles. That will be very hard and it will involve not just education and language support, but better libraries, better analysis tools, and tools that support code rejuvenation/modernization. This is not a task that can safely and economically be done by hand.

• Programming may be really addictive, but there may be also some problems with burnout. If we could ask, what are your ways of rest from programming, and the whole IT?
  o Running, reading, traveling, hiking, children, listening to music, friends.