Follow-up Questions and Answers after the PLDI 2020 AMA of Bjarne Stroustrup hosted by John Regehr:

1. I'm going to ask this as half of a joke, so please take it lightly. In the mid 2010s, a bogus story went around that "you created C++ to raise developer salaries." While the story turned out to be false, developer experience of C++ has quite a bit to be desired with regards to memory safety, number of keywords, etc. Instead of asking if you regret any features of C++, in hindsight, are there features that you envy in other languages? Some languages have static checks built in (Rust), others have very minimal syntax (Haskell, Lisp).
Asked by - Yoshiki Takashima, PhD Student @ CMU

⇨ Actually, that fake interview is from the 1990s. By now, I find it rather tedious and it never bore much resemblance to reality outside people who had been badly taught. Unfortunately, there were and are many of those. IEEE asked me about it in 1998: https://www.stroustrup.com/ieee_interview.html .

⇨ I think that the major design decisions for C++ were reasonable, and still are. Just about every detail could – in retrospect – have been done better. For my views on the history of C++, see my History of Programming language papers, especially the one from this year (2020): Bjarne Stroustrup: Thriving in a crowded and changing world: C++ 2006-2020. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. London. June 2020.

⇨ For complete type-safety and resource safety, I look to the C++ Core Guidelines and developments of those. We already do far better than most people are aware of.

2. Could you give some comments on WebAssembly? Do you think it will achieve success in non-web applications?
Asked by - Wenwen Wang, University of Georgia

 I like the idea of WebAssembly, but I am not an expert. I have no idea if there is a market for it outside the Web.

3. Compilers have indeed gotten excellent at optimizing C++, but doesn't this come at the cost of performance predictability? In my research writing hardware simulators in C++, I've seen 10x speedups or slowdowns from trivial changes (typically small changes that break inlining heuristics). How do we get high-level semantics and predictable performance?
Asked by - Clément Pit-Claudel, MIT CSAIL

 A hard question. Modern optimizers are uncomfortably close to "black magic." However, I increasingly find myself relying on it to allow me to write cleaner, simpler, higher-level code. I don't want to lose that and for performance we need careful testing whichever way we write our code. For example, code that is performant in isolation can be far worse (e.g., your 10x or much more) when embedded in a large system.

I think we can get closer to "high-level semantics and predictable performance" through a combination of better optimizers and good coding guidelines, but unless we slow down

our code significantly, we will still be subject to the vagaries of cashes, pipelines, dynamic reordering, and branch predictors.

4. Are you interested in what's happening with language design outside C++, and do you try to "steal" some niceties from other languages for newer editions of C++? Could you name some examples? Especially in the field of type systems features, if any.
Asked by - Artem Pelenitsyn, Northeastern University

⇨ I'm interested, but I never steal. I am quite scrupulous with acknowledgements. Most novel features in all languages emerge from a variety of research sources, rather than from a single major language. I don't get much inspiration from type theory. In general, I find theory and language research better sources of solutions than as inspiration for what problems are worth solving. The best source of insight into what needs to be done is real-world application. Remember Kristen Nygaard's dictum: "If we design languages so that you need a PhD from MIT to use them, we have failed." I aim to build a practical tool for millions.

5. What are your thoughts about Python centric development, while using C++ in places where performance matters? Can this approach be successfully used to develop large and complex systems?
Asked by - Anshuman Dhuliya, IIT Bombay

 Python is a most useful and powerful tool, especially for experimentation, prototyping, scripting, etc. By using something like C++ for the performance-critical parts extends the range of uses for which this approach is viable. However, I don't think that dynamic typing plus testing is suitable for large, long-lived, safety-critical code bases under constant maintenance. For such systems, I prefer to rely on static type safety. I also favor a more deliberate design approach than I most commonly find with Python.

6. You mentioned better support for distributed programs as one of your future goals. In what ways are the current distributed programming interfaces like MPI inadequate and where should the improvement effort be focused?
Asked by - Peeyush Kushwaha, IIIT-Delhi

 I think MPI is basically an assembly-level mechanism. If you write it by hand your code is brittle and its performance suffer each time you change your hardware. We need to develop higher-level models that can be automatically tuned at run-time and generated to suit a variety of hardware and configurations.

7. What do you think about dependent type? Is it possible to extend C++ toward a stronger static

type system?
Asked by - Chao-Hong Chen, Indiana University

⇨ We will see – are seeing – strongly typed C++, but that is not because of adoption of novel type systems. Dennis Richie famously said "C is a strongly typed, weakly checked programming language." I take the same approach and rely on a combination of coding rules, simple support libraries, and static analysis to achieve completely type-safe and resource-safe C++. For example, see A brief introduction to C++'s model for type- and resource-safety . Support for that ships with Visual Studio. Compatibility constraints and scaling concerns prevent purely language-based approaches from succeeding.
⇨

8. Research advances the state-of-the-art. With all the advances coming from research, what excites you most coming from research, and do you see it being suitable for inclusion in a future iteration of C++?
Asked by - Jan de Muijnck-Hughes, University of Glasgow

 The snag is that it is very hard to guess which novel research ideas will actually work at scale under real-world constraints. I have seen too many bright and clever ideas not work out to get easily excited. Currently, I am more interested in an engineering approach integrating what is known to work (somewhere) and to simplify the use of C++ for less expert users than in the latest research.

9. A follow-up to the performance predictability question: Are there ways in which C++ and other programming languages can still get closer to the hardware and allow for more control? Particularly thinking in the context of performance optimization for modern computer architectures or security given microarchitectural side channel attacks possible on modern CPUs with programmers desiring guaranteed preservation of constant-time properties of cryptographic code (or, would that last domain still be one area where architecture-specific assembly code remains legitimately the best practical choice)?
Asked by - Matt P. Dziubinski

- Hi, Matt! I think that we must think in terms of managing complexity through layering. We need high-level, clean, simple, type-safe interfaces to our key facilities. Examples include the standard-library parallel algorithms. Often, high-level optimizations are feasible at this level and harder at lower, less abstract levels, such as code written using traditional C-style loops. When we need to, we can peel of a layer to gain more detailed control. As we peel off layer after layer, we gain more control and often access to specific machine facilities that the compilers and optimizers doesn't (yet) know how to use. We also lose generality of our optimizations and have to write more and more tricky code, opening ourselves up to more errors. That's why I call this "the onion principle" as you peel off more layers, you cry more.
- For type-safe performant code that doesn't need access to specific hardware properties, I tend first to look towards compile-time programming (**constexpr**) and may even consider in-line assembler (**asm**).

10. How long did you spend on designing C++ before you implemented the first working version of it? How long did it take you to implement the first version?
Asked by - Haipeng Cai, Washington State

 I started by designing around a few good ideas – notably simple classes, constructor/destructor pairs for general resource management, and function argument declaration and checking – and implementing as I went along. I had my first non-research user after 6 months. After 5 years (in 1985), I had a refined, reimplemented, and stable version that could be widely used commercially and in academia. Basically, from 1979 to about 1989, I was doing full-time simultaneous/concurrent design, implementation, library building, user support, application building, documentation, and teaching. This is described in my book "The Design and Evolution of C++" from 1994 and in my first HOPL paper: A History of C++: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28 No 3, pp 271-298. March 1993.

I think that it has been crucial for C++'s success that it has grown incrementally based on feedback from wide use. That makes for messy details and compatibility constraints – stability id an important feature – but help avoid major mistakes.

11. Is there anything you would like to see in LLVM (from the perspective of C++) and do you have any opinions on MLIR?
Asked by - Klas Segeljakt, KTH Royal Institute of Technology

⇨ Better support for static analysis and in particular complete support for the C++ Core Guidelines .
Sorry, I don't know MLIR well enough to have an opinion.

12. Since C++20 introduced modern formatting facility std::format , C++ "hello, world" needs an update. Would you update The C++ Programming Language to use that, and/or encourage other book authors to do so?
Asked by - Zhihao Yuan, SimpleRose Inc

 A printf-style formatting sub-language is "modern formatting"? Well, many people like it and can now use it. I have no current plans for a 5th edition of "The C++ Programming Language." Writing that would be a massive undertaking. I will have to consider your question when I revise "A Tour of C++" next year when I know more about what works best in C++20 and how, but that answer is not trivial. I/O streams work rather nicely with my uses and **<format>** integrates nicely with **iostream**s. In particular, it does not compromise iostream's type safety.

And, no, I don't tell other book authors what to do.

13. Where would you recommend to read and learn about memory models and memory order in general and of C++ in particular?
Asked by - Gal Milman, Technion

- There are two levels of understanding of memory models and therefore two answer to your question:
  - For most programmers: Thanks to a contract between compiler writers and machine architects, memory in C++ works roughly the way you expect it to. If you are in a concurrent system, you need to avoid race conditions. This is done by not sharing data (preferably) or by making sure that access to shared data is controlled. The C++ standard library offers atomic types (e.g. **atomic<int>**), locks (e.g., **scoped_lock(m1,m2)**), and more for that.
  - For concurrency experts (quoting from my HOPL-4 paper): Basically, the C++11 model is based on happens-before relations [Lamport 1978] and supports relaxed memory models as well as sequentially consistent [Lamport 1979] ones. For details, see Paul E. McKenney, Mark Batty, Clark Nelson, Hans Boehm, Anthony Williams, Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber, Michael Wong, Lawrence Crowl, and Benjamin Kosnik. 2010. Omnibus Memory Model and Atomics Paper.ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3196. 11 Nov. 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3196.htm (also at Internet Archive 4 Sept. 2019 02:05:46). Much is based on a formalization by Mark Batty and others at the University of Cambridge: Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In Proceedings of the40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Rome, Italy, Jan.) (POPL '13).Association for Computing Machinery, New York, NY, USA, 235ś248. 978-1450318327https://doi.org/10.1145/2429069.2429099 .