The raw text of Owen Hughes interview for TechRepublic C++ programming language: How it became the invisible foundation for everything, and what's next November 2020.

- What was your initial motivation behind designing C++?
  - I wanted to build a multi-computer system with a communication system that could be either shared-memory or a network. My focus was on the software. I needed
    - To write low-level, close-to-hardware code, such as memory managers, process schedulers, and device drivers.
    - To separate software components so that they could be running on separate computers communicating in well-defined ways.
  - No language could do both, so I had to build something that could. "C with Classes" (as C++ was initially called) combined C's ability to work close to the hardware with an efficient variant of Simula's classes for abstraction and code organization.

- What do you see as the biggest milestones for C++ over the course of its ~35-year history?
  - In 1979, during the first month of my work on C++ (then called "C with Classes"), I added function prototypes and classes with constructors and destructors.
  - In the fall of 1985, the first C++ compiler (Cfront) and the first C++ book ("The C++ Programming Language") were released on the same day.
  - In 1989, the ANSI (later ISO) standards effort started – initiated by IBM, HP, and Sun.
  - In 1994, Alex Stepanov's framework of iterators, algorithms, and containers (the STL) was accepted.
  - In 1998, the first ISO standard was issued including templates and exceptions. In the years following, C++98 became a solid workhorse.
  - C++11 made C++ feel like a new language. The type-safe support for concurrency was essential. C++11 supplied a dense web of mutually supporting features such as constexpr functions for compile-time computation, lambdas, auto for type deduction, and variadic templates. It laid a solid foundation for future evolution.
  - C++20 comes close to meeting my aims for C++ (as articulated in "The Design and Evolution" in 1994) with modules for better code hygiene and much faster compilation, concepts for better generic code, coroutines for

more flexible order of execution, and improved support for compile-time computation.

For "details" of C++'s design and Evolution see my HOPL (ACM SIGPLA History of Programming Languages) papers, especially the most recent one: Thriving in a crowded and changing world: C++ 2006-2020.

- How much has the language evolved since its inception – would the original design still be recognizable today?

  - The original design is very visible today. There are simple programs from the early years – 40 years ago – that would still run today. Stability is an important feature for a language used for systems that has to work for decades. In fact, many of the early ideas (e.g., as documented in "The Design and Evolution of C++" from 1994) became available only in C++20.
  - Today's C++ is of course far more powerful and expressive than the early C++. I knew from the start that I couldn't build the ideal language, so I had to aim for gradual development – evolution. In fact, I did not believe in the idea of a perfect language – perfect for what? For whom? Evolution is necessary to meet the challenges of a changing world and to incorporate new ideas.

- Where do you see C++ in terms of its place in the current developer landscape? Where are we seeing most commonly used; similarly, are there places where it's seeing increased/ decreased use?
  - If you have a problem that requires efficient use of hardware and also to handle significant complexity, C++ is an obvious candidate. If you don't have both needs either a low-level efficient language or a high-level wasteful language will do. This was understood from day #1.
  - C++'s abstraction mechanisms allow us to meet safety requirements. If you want type- and resource-safe code, you can do that in C++ without significant cost. The key here is to enforce modern C++ design and programming techniques. The C++ Core Guidelines aims at that and to offer enforcement using static analysis. The analyser shipped with Microsoft Visual Studio offer protection against memory corruption and resource leaks; other analysers are offering increasing numbers of

guarantees. We need to distinguish between what can be done (according to the ISO standard) and what makes sense (is safe and efficient).
  - It is extremely hard to determine where C++ is used and for what. A first estimate for both questions is "everywhere." In any large system, you typically find C++ in the lower-level and performance-critical parts. Such parts of a system are often not seen by end-users or even by developers of other parts of the system, so I sometimes refer to C++ as "an invisible foundation of everything." Counting programmers is hard and simple Web surveys typically just measure "noise"; that is what is being talked about as opposed to what is being used. Surveys (e.g., the JetBrains one) show a C++ user population of at least 4.5 million with a steady growth o about 100,000 developers a year.

- How does C++ account for being such a popular programming language? What have been the major design and use factors that have got it to where it is today?
  - C++'s success was obviously a surprise. A language without rich sponsors, serious marketing, or a development centre is not supposed to succeed on a large scale, but C++ did so over four decades.
  - I see C++'s success as a function of its original design aims (efficient use of hardware plus powerful abstraction mechanisms) and its careful evolution based on feedback from real-world use. If I should single out language features, it would be
    - classes with constructors and destructors
    - templates (now with concepts for expressing requirements)
  - It was essential that the evolutionary strategy emphasized stability and compatibility.

- How does one lead the development of an entire programming language? What are the challenges that go with a project so large and how does one manage them?

  - You start small, articulate fundamental principles, articulate long-term ideals, and develop based on feedback from real-world use (sticking to the principles and ideals).
  - From the earliest days, I realized that I didn't have (dictatorial) control of the language, only influence. Once you have users who depend on your

work, you are responsible. Only through paying attention, careful thought, and hard work can you contribute constructively.
    - Over the years many people contributed to C++ (the language and the standard library). In the beginning, it was just a few colleagues at Bell Labs, then dozens of people in the standard committee, and now on the order of 400 people in the standards committee (there were 252 people at the most recent face-to-face meeting in Prague where we approved C++20) plus a wider community that pays attention to C++'s evolution and tries to influence it. That's an opportunity and a huge problem. My recent HOPL paper has a discussion of the difficulties of keeping a language coherent given so much enthusiasm.
    - The hardest part is to decide what's important and maintain a coherency. Once you know what you want, eventually, you find a good technical way of doing it.

- Similarly: when designing a programming language, how do you reach a consensus on deciding which new features to adopt and omit?
    - Through lots of hard work and discussion. This takes time and patience.
    - You must try to add only what really helps people and then only a few such things because if we accepted every feature that would help someone, the language would sink under its own weight. We can't accept even all good features.
    - I remind people of the Vasa, the beautiful 17th century Swedish battleship that sank in Stockholm harbour on its maiden voyage. At the insistence of the King (highest management!) and against the better judgement of the technical people, it had been piled high with beautiful statues and great guns. Top heavy, it was overturned by a gust of wind. I repeatedly talked and wrote about the about the Vasa as a caution to people enthusiastically wanting to improve C++ by adding features: Remember the Vasa! So far, C++ hasn't tumbled over.

- C++ is considered a somewhat more challenging programming language for newcomers to get to grips with. Do you think this is a fair assumption? What features have been added in recent years to make it more accessible?
    - C++ is indeed complex and it takes effort to learn to use it well. Unfortunately, people don't just want simplicity, they what something

impossible: a simpler language, with more features, and no breakage of their existing code.

- My approach to that "trilemma" is to
  - add features to make simple things simple to do (e.g., though generalization or direct support for common cases)
  - maintain compatibility/stability
  - articulate rules for using modern C++ well and ways of enforcing those rules.
- For example, a range-for loop, e.g.,
  - for (auto& x : vec) x=0;
- is simpler and gives less opportunities for errors than a traditional C-style loop, e.g.,
  - for (int i=0; i<vec.size(); ++i) vec[i]=0;
- Another favourite example is to avoid verbosity and leaks by using local objects rather than direct use of new:
  - void poor(int i) { X* p = new X(i); … delete p; }
  - void better(int i) { X x(i) … }
- If you really need a pointer, you can use a smart pointer:
  - void if_needed(int i) { auto p = make_shared<X>(i); … }
- We really don't need to write verbose, error-prone, old-style C++ any more.

- How much direct involvement do you have in the development of C++ today?
  - I'm involved in quite a few things:
    - I'm part of the direction group, discussing and presenting recommendations about the future of C++: Direction for ISO C++.
    - I follow the evolution group and take part in discussions about new language features (such as unified function calls, operator dot, contracts, exceptions, static reflection, and functional-style pattern matching) and to a lesser extent other groups such as library and education.
    - I follow administrative activities, but try to do as little as possible there; I am not a great administrator.
  - Naturally, many things are different this year with meetings cancelled and/or moved to the Web because of the virus. I find that difficult and it is slowing up much-needed work.

  - I do my bit to explain C++ to the world at large through my books, articles, videos, and interviews. Before the virus, I also travelled a lot to learn and teach.

- What is the nature of your current role at Morgan Stanley, and how do you juggle that with your position as Honorary Doctor at Universidad Carlos III de Madrid?

  - At Morgan Stanley, I'm a Technical Fellow. There, I mostly deal with distributed systems, programming techniques, and teaching. My work with the ISO C++ standard and on the C++ Core Guidelines are considered part of my job there.
  - My Carlos III doctorate is honorary, so there are no formal obligations, though if it wasn't for the virus, I would have visited them this year to give a talk and meet with people, just as I did last year. Every year, I give software design course using C++ at Columbia University in New York City.

- Your students have reserved you a Twitter page – what's the story there? Have you used it yet, or do they manage your social media activity?

  - I lurk. I see what's going on and only very occasionally respond. I try to limit my activities to my papers, talks, and interviews. I dislike the rapid-fire exchanges on social media and prefer the more thoughtful and considered forms of communication.
  - I don't currently have graduate students and I don't have my communication managed or filtered by anyone. Of course, I often try out ideas with friends and colleagues before making them more widely available. I appreciate it when someone takes an action to protect me as a student did when he grabbed @stroustrup for me before I knew about twitter.

- We see programming as being a very sought-after skill in the current climate in particular – how do you see the role of software engineers evolving in the next few years as more companies go digital, as well as with the rise of low-code/ no-code solutions?
  - There are many kinds of programming. Some are relatively simple and can be done by essentially everyone. Other kinds are highly specialized and

require skilled experts and experienced engineers. Most are in-between these extremes. I fear that the various kinds of tasks and level of expertise are often confused. There really is a vast difference between setting up a simple web site and building the infrastructure for a service on which lives or livelihoods critically depends. I am primarily interested in the latter: how it is done, what tools are used, and how the experts are educated.

- I think what we will see is that "digital" will become an immensely varied field with an extremely broad range of tasks and skills. It is not simply a linear progression from the simple to the complex. Especially at the expert level, there are degrees of specialization that requires many years of experience at the expert levels.

- The issue of developer burnout is a big topic right now. What do you see as being the main challenges for the developers of today, and what can IT managers/ CIOs do to support them better?

  - Actually, I suffer from a bit of burnout myself. For my work, I depend critically on talking with people to learn about their problems and hear how my ideas might help them. In this time of the pandemic, I am deprived of much-needed feedback. "Virtual" talks and interviews are not the same, and the dynamic of Zoom meetings are inferior to real face-to-face meetings when it comes to discussing design and ideas. To make matters worse, every organization tries to keep active and relevant by adding meetings. Some do it to survive.
  - What can managers and executives do to help? Hard to say in general, but in many cases doing less would help. Know "your people" and interact with them in ways that suit and help individuals – don't generalize and formalize. Keep an eye out to ensure that no one is left isolated unless they really want to be left alone. Encourage work that can be done in isolation and by small groups. Leave meetings and "virtual events" to informal initiatives. After all, it is the informal contacts and interactions that most people feel lacking. Support and encourage such informal activities, but don't add to the burden of many meetings.
  - In the software industry we are lucky to be able to work remotely, rather than taking the risks of daily constant interactions with potentially infected people. We should not complain too much.

- What are the main features we can we expect in C++20 and C++23, and where do you hope to see C++ going beyond this?
  - We (the ISO standards committee) have a plan:

    ”for C++23, let's work towards having the following things in that standard:
    Library support for coroutines
    Executors
    Networking
    A modular standard library
    Without a particular ship vehicle yet, we should also make progress on
    Reflection
    Pattern matching
    Contracts”

    Given the upsets and obstacles caused by the pandemic, it is unlikely we will get more than one or two of those features as early as 2023, but we try our best. Beyond that, there is work on Unicode, numerics, game development and low latency, tooling, AI, and much more.

  - We ship a feature (language and library) when it is ready and we issue a revised standard every 3 years. C++14, C++17, and C++20 shipped on time. It is worth noting that the standards effort and the major implementors are very much in sync: almost all of C++20 is shipping in 2020.
  - It is crucial that C++ remains coherent and is a stable platform for development.
  - I recommend my recent HOPL paper as a far more detailed discussion of C++’s design, evolution, and use. It also has a host of code examples to illustrate design decisions and specific language features.