Raw interview with Jagmeet Singh for OSFY Magazine. April 2017.

- How did you get the idea to develop C++?

  I needed a language that could be used to manipulate hardware directly and use all the available hardware resources well (C was an example). I also needed a language that allowed me to handle complexity (Simula was an example). There wasn't a language that could do both, so I started to build one by adding Simula's class ideas to C.

- For whom did you build the initial C++ model?

  I wanted that language for myself to help build a distributed system based on Unix. Before I even finished my language, my friends and colleagues at Bell Labs started to use it for their projects, often simulation projects because the very first library I built for "C with Classes" (as I called my language) was a co-routine library. So, I built C++ primarily for myself and my colleagues. However, the range of projects, demands, and hardware at Bell Labs were very wide so the language had to be very flexible to cope with that.

- What were the prime challenges you faced in inventing C++?

  There were many challenges because I was building a tool for practical use, rather than as an academic project for publication. A tool has to be good enough for everything its users need; just being the best in the world for one or two things is not sufficient for success.

  - Language design: what features do I and my colleagues need to simplify our code?
  - Implementation techniques: How do you implement those features so that they are affordable to use in real-world development and execution environments?
  - Installation and portability: How do you get the new language to run on a variety of hardware and operating systems?
  - Education: How do people learn to use the new technique and language features?

  I wrote up answers to such questions in my ACM History of Programming Languages papers (http://www.stroustrup.com/hopl2.pdf, http://www.stroustrup.com/hopl-almost-final.pdf) and in my book "The Design and Evolution of C++" (Addison-Wesley). It wasn't all that easy because for the first years I was the only person working on "C with Classes." My colleagues were most supportive, but there wasn't an official C++ project with a budget; basically, the help I offered to a range of Bell Labs projects allowed me to develop C++.

  There were of course many languages competing for programmers' attention at the time, such as Ada, C, Lisp, Modula-2, Smalltalk, Objective-C, Pascal, and ML. I

looked at those, and many more for ideas, but mostly ignored them as competitors. I did not have any resources to use in a commercial competition, so I simply stuck to making C++ better for the tasks I and my colleagues needed it for. That is still my basic policy vis a vis other languages.

- Why was there a need for C++ when C already existed in the computing world?

  I built C++ on C, because I did not want to build from scratch; that would not have given me a useful tool in a reasonable time. However, C could not and still cannot manage complexity as well as C++. C and the use of C has evolved over the years – often under the influence of C++. When I started with "C with Classes," the use of sets of functions as opaque interfaces (which today is C's most effective abstraction mechanism) wasn't common. It has been conjectured that this developed partly in response to C++'s classes and virtual functions. K&R C did not offer function prototypes, //, const, inline, and more; those came from my work with C++. Note that even GCC is a C++ program now.

- How has been the evolution of C++ in the programming space?

  I'm not sure I understand your question, but let me try. For the first 10 years or so C++'s user population doubled every 7.5 months. Currently, there is about 4.5 million C++ users and that number is growing by about 100,000 a year. In 2015, the people building the Clion C++ IDE made a survey: https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/ . C++ use is all over the world and is heavily used in finance, banking, games, front office, telecoms, electronics, marketing, manufacturing, and retail. From my own experience, I can add embedded systems, scientific computation, and graphics.

- Is it the open source practice that brought early success for programming languages such as C and C++?

  No. The early success of C and C++ predates the emergence of open source. However, AT&T did the next best thing and allowed use of C and C++ compilers and libraries for a very low price. For non-profit organizations, C++ cost $75, which was the cost of the magnetic tape on which it was shipped (source and binary); organized distribution over the Internet was still in the future. Soon, AT&T gave the specifications of C and C++ to the ISO so everyone could use them, and I personally helped other organizations with getting C++ compilers written. Nowadays there are of course open-source and proprietary C++ implementations. To ensure a wide reach of my work, I deliberately (with the agreement of AT&T) refrained from patenting anything related to C++.

- Do you see any programming languages today that can replace C++? Or is it irreplaceable?

I don’t see a current language that could replace C++ across its range of uses; its combination of hardware access and zero-overhead abstraction is still unique. However, nothing lasts forever. Eventually, a current language will acquire sufficient facilities or a new language will come along. So far, C++ has been at the top of the game for almost 30 years. That’s not bad, especially given that C++ never had a powerful owner or a marketing budget.

- What are the major features that make C++ an easier option against C?

  The better type system, classes with constructors and destructors, overloading, support for object-oriented programming, support for generic programming, support for compile-time programming, and in many areas exception handling.

- Why should aspiring developers need to learn C++ to survive in the growing world of computers?

  I don’t know if they need to learn C++, but they should want to: It’s one of the most widely used languages. It’s one of the most flexible languages. It’s one of the languages that delivers the best performance. It’s one of the languages that allows you direct access to hardware resources. It’s one of the few languages that allows you to use a wide range of fundamental programming techniques. It’s a language that allows you to work in most industries.

- There are different compilers available for C++. Which one is the best pick in your view and why?

  I don’t have a favorite. I use several. The major C++ compilers are all good. They have good standard conformance, generate good code, and have good supporting toolsets and fundamental libraries. You choose a C++ compiler based on specific needs, such as ability to use specific hardware (e.g., the IBM compiler provides good support for the PowerPC), specific programming techniques (e.g., the Intel compilers have good support for vectorization), specific environments (the Microsoft compiler provides good Windows support), portability (GCC or Clang), or a specific toolset (e.g., GDB or Visual Studio).

- Unlike Python and Java, C++ has not yet been a perfect choice for embedded engineers. Do you think the focus should now largely be expanded towards connected devices?

  Python and Java are certainly not perfect choices for embedded systems either. C++ is critical when you need to squeeze performance or energy efficiency (e.g., battery life) out of a gadget, but there is no one language that’s best for everything and everybody. There is a lot of C++ embedded system code. It is worth remembering that “embedded systems” is a huge range of things from coffee machines to jet-plane flight controls, from fuel injectors to stereo amplifies, from lithium-ion battery controllers to self-driving cars, etc. There is a corresponding range of needs for programming techniques and tools.

- How do you perceive the importance of readability in a programming language?

  Readability is essential. If you can't read code, you can't maintain code and you can't reason about correctness. Today's C++ can be so much more readable that older C++ or C. I'm working on an ambitious project to define what modern C++ code using C++11, C++14, C++17, and beyond should look like. It is called the Core Guidelines (https://github.com/isocpp/CppCoreGuidelines) and is an open source projects with editors from Morgan Stanley, Microsoft, RedHat, and FaceBook. We aim for completely type-safe and resource-safe code, without limitations on expressibility or performance. Code conforming to the Core Guidelines is far more regular and readable than most current code. We are working on tools for automatic enforcement of those guidelines.

- Do you see any big difference between computer systems and embedded devices that are designed for the Internet of Things (IoT) ecosystem?

  There are differences, such as ABIs and security concerns, but the fundamental programming needs and constraints are the same, and I think they favor C++.

- Why do you think the world is shifting from low-level languages to high-level, interpreted languages?

  First, I would not equate "interpreted" with "high level." Interpreted code can be as hackish as the lowest-level C code. Second, I don't think the world is shifting like that. What we see in an increasing fraction of programming becoming routine and relatively undemanding of programmer skills and not under serious resource or performance constraints. The amount of work that is not like that (resource intensive, safety critical, close to the hardware, etc.) and require serious programming skills is still increasing, but not as fast as the code that does not have as stringent requirements. For example, we are seeing an increased trend towards "native code" in smartphones even as the amount of non-native (hosted, interpreted, etc.) code is going up fast.

  It is worth remembering that the major "engines" and VMs for the interpreted languages are C++ programs.

- How do you plan the revisions of C++ standards?

  We just moved C++17 out for national vote. It will be the ISO C++ standard later this year. C++20 will be the standard after than in 2020. For C++20, I cope for concepts, modules, and possibly contracts and coroutines. It could be an exciting major revision and change the way we program, but of course all depends on the committee's willingness to accept change.

  Concepts are shipping in GCC. Modules are shipping in Microsoft. Co-routines are available in Microsoft and Clang. More implementations are in the works, so my

dreams have some concrete foundation. You can download compilers and try out the features today. You can also try the major compilers on-line: https://gcc.godbolt.org/ .

It is extremely difficult for a group of 200 to make plans and stick to them. Some of us in the committee try, though. For example, see a note from a group of heads of national standards bodies presented in Kona: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0559r0.pdf . They ask for better direction of the committee's efforts.

- What are all the scheduled features in the C++17 standard that will advance the present programming experience?

  Compared to C++11 and C++14, C++17 doesn't offer anything that will fundamentally change the way you program; it doesn't change the way you think about constructing a program. Thus, by my usual definition, it is a minor update. It does, however, offer a little for everybody. Importantly, most C++17 features are already shipping in the major implementations; I suspect the major implementations will all be feature complete before 2017 is out. So, C++17 in 2017!

  Lists are easy and boring. You can find them all over the Web (e.g., Wikipedia). Here is one:

  - Language
    - Structured bindings. E.g., **auto [re,im] = complex_algo(z);**
    - Deduction of template arguments. E.g., **pair p {2, "Hello!"s};**
    - More guaranteed order of evaluation. E.g., **m[0] = m.size();**
    - Guaranteed copy elision
    - Auto of a single initialize deduces to that initializer. E.g., **auto x {expr};**
    - Compile-time if, e.g., **if constexpr(f(x))** …
    - Deduced type of value template argument. E.g., **template<auto T>** …
    - **if** and **switch** with initializer. E.g., **if (X x = f(y); x)** …
    - Dynamic memory allocation for over-aligned data
    - **inline** variables (Yuck!)
    - **[[fallthrough]]**, **[[nodiscard]]**, **[[maybe unused]]**
    - Lambda capture of *this. E.g. **[=,tmp=*this]** …
    - Fold expressions for parameter packs. E.g., **auto sum = (args + …);**
    - Generalized initializer lists
    - …
  - Standard Library:
    - **variant**, **optional**, **any**, **string_view**
    - File system library
    - Parallelism library
    - Special math functions. E.g., **riemann_zeta()**
    - Many minor standard-library improvements
    - …

Depending on the level of detail, you can get a longer or a shorter list. What matters is how these features can be used to write better code. By "better" I mean code that more directly expresses its intent and runs faster using fewer resources. The list above is ordered in a rough order of importance. Note in particular "structured bindings" that allows us to more directly use multiple values returned from a function. For example, here is how we can print a map of key-value pairs:

```
for (const auto& [key, value] : mymap)
        cout << key << " -> " << value << '\n';
```

The new stricter order of evaluation rule (left to right except for assignment which is right-to-left (just like the grammar) is invisible, but prevent bugs:

```
int main()
{
        std::map<int, int> m;
        m[0] = m.size();          // What is the value of m[0]?
}
```

Since the dawn of time, the order of evaluation of **m[0]** and **m.size()** has been implementation defined and a source of confusion of bugs. Now, the result is 0 because the size if taken before the insertion.

- Spending hours on the code is not an easy task for young programmers. What valuable tips can they get from you to comfortably work on new projects?

  But it is necessary time spent. Code is where ideas turn into practical results. We can't just write papers, manuals, etc. We must produce code that actually works.

  Before adding to a code base, you have to understand some of it. You can get a general understanding of some code top-down, but to be able to make a change (e.g., to fix a bug) you need to understand the code inside-out: what affects this line of code and what does this line of code affect? Anything that helps read code and navigate through code helps. Linear reading or trying to understand everything just don't work. It is infeasible in large code bases and inefficient in small ones.

  So, look for good code navigation tools (e.g. IDEs) and of course hope for helpful colleagues who will spend time introducing you to the code base and its associated tools (debuggers, build systems, test suites, etc.).

  If you want to use C++, take care to learn it well. In particular, learn to use modern C++ well. Don't just repeat the errors of the past. You can write so much better C++11 than you could write in the styles of the 1990s. To help, there are the Core Guidelines (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md).

If you are a programmer needing to brush up on modern C++, I can recommend my nice thin (190 page) book “A Tour of C++” for a quick overview. What people often miss about modern C++ is not the individual features (as you find them in lists), but the way those features can be used in combination to support better programming styles.

- Lastly, where do you see the programming world in the coming future?

  Hopefully, it will be easier to read, write, and maintain code. I don’t want to write science fiction so I won’t go into details, but I expect that I and C++ will make a significant positive contribution to that future.