Questions from Paul Krill from Infoworld; answers from Bjarne Stroustrup

The edited and abbreviated for publication version is here.

* When will C++ 17 be available?

C++17 will become official sometime in 2017, probably in the fall, and the major implementations are likely to be ready then, or even before. Parts are shipping already.

* What do you see as the major new features?

Define “major”. I consider a language feature or a library component major if it affects the way you think about programming and affects how you structure your code. With that definition, sadly, my answer must be: For most people, I don’t see anything major in C++17.

I like the file system library and the parallel algorithms. They are useful and will make some tasks easier for many, but I don’t consider them major.

However, many of the features that I consider major are available in some form or other today. Have a look at Herb Sutter’s recent blogs about the Jacksonville meeting and how the committee operates:

- https://isocpp.org/blog/2016/03/trip-report-jax-sutter
- https://isocpp.org/std/the-life-of-an-iso-proposal

In addition to the standard itself, the committee produces Technical Specifications (TSs) and members of the community involved with the committee produce implementations. The major features are appearing as TSs. For example:

- Concepts
- Networking
- More concurrency stuff
- Ranges (STL2)
- Modules
- Coroutines

Can we say that these are “part of C++17”? Not really; they will not be part of every C++ implementation, but they exist. They are backed by committee votes. They are carefully documented and usually their implementations have gone through a few revisions. A TS, once approved and issued by the ISO (as for example the Concepts TS has been), has official standing.

Herb Sutter (the C++ standard committee’s convener) and others are encouraging us to consider these TS as “beta releases” of standard facilities. I don’t know how far we should push that analogy, but if you are willing to use a beta release of some software, you should consider these features.

* With the addition of constexpr lambdas, does C++ continue to become more of a functional programming language? What does that mean for C++ developers?

Since the introduction of the STL (about 1994) there has been a steady and cautious increase in the use of functional-programming techniques in C++.

Constexpr lambdas is simply an extension of the set of features that can be used at compile time, rather than something specifically functional. You can now also have loops in constexpr functions (and through that in constant expressions).

If the “structured bindings” proposal is accepted for C++17, functions with multiple return values will become easier to use, much as such functions are used in functional programming.

* Apparently, Concepts, for improving compiler diagnostics, won’t make it into C++ 17 after also not making it into previous releases. Is this a big disappointment?

Yes, for me at least, it this a huge disappointment. Together with Gabriel Dos Reis and others, I have worked on this problem for a couple of decades, on this particular approach since 2003, and we have had Andrew Sutton’s implementation to play with for about 3 years. I consider it ready for a standard release next year, but a large number of committee members disagreed (for various reasons).

Very soon, concepts will ship as part of GCC6.0, so by the time C++17 ships next year, the standard will have to play catchup.

Please note that the current concept design represents a completely different approach from the failed C++0x concepts. The current approach is simpler and in many ways more powerful and flexible. A concept is simply a compile-time predicate on a set of types and values.

I consider “better error messages” a (most useful) consequence of the fundamental advantage of concepts: we can specify the requirements of our generic code (templates) on its arguments. This leads to better designs, better interfaces, the ability to use simple overloading, simpler implementations, and potentially more efficient code (through simpler code).

Concepts will do for generic code what function argument declarations (function prototypes) did for ordinary code using K&R-C-style functions. Today, we have a really hard time imagining how people managed before that (1979 for C++, earlier for many other languages). In a few short years, we will think the same about concepts.

* Can you answer the same question about Modules and Coroutines, which also won’t make it in?

I would have liked to get modules for better protection against changes in some components context (e.g., protection against macros) and better compiler speed, but that proposal isn’t ready for C++17, so it goes into a TS.

I think that eventually, modules will become massively important. They address long-standing problems in C and C++. An early version currently ships as part of Microsoft’s C++ compiler. A different variant, more

dependent on external tooling and more friendly to macros, is available in some versions of Clang.

I am disappointed that stackless co-routines are being put into a TS rather than directly into the standard itself. I think they are ready and important for a few critical use cases (pipelines and generators). An early version currently ships as part of Microsoft's C++ compiler.

* Why didn't you just delay shipping the standard for a year and get concepts, modules, and coroutines?

I was asked that question directly in the plenary session. My answer was roughly: No, we must ship C++17 as promised. A delay will set a very bad precedence and cause more delays in the future. If C++17 became C++18, I suspect that C++20 would become C++22 or C++23 and we would be well on our way back to the 10 year cycle for ISO standards.

* is this an adequate explanation of coroutines for C++'s purpose? Coroutines are computer program components that generalize subroutines for nonpreemptive multitasking.

No. I don't think those references help; this is better. Think of a coroutine as simply a function that resumes from where it returned the last time it was called. For example, we can write a naïve (and efficient) coroutine to generate Fibonacci sequence like this:

```
gen<int> fibonacci() { // generate 0,1,1,2,3,5,8,13 …
        int a = 0;     // initial values
        int b = 1;

        while (true) {
                co_yield a;    // return next Fibonacci number
                int next = a+b;
                a = b;         // update values
                b = next;
        }
}
```

The co_yield statement returns a value and waits for the next call. We could use it like this:

```
for (auto v: fibonacci()) cout << v << '\n';
```

Note I use no explicit global state and no static variable. The point is that for realistic examples keeping the state of a computation is non-trivial and coroutines handle that for you. The code generate for this fibonacci() is as efficient as any hand-optimized version using a function and some non-local state.

These are stackless coroutines. That is, you cannot suspend and resume in a function called from the coroutine. That's the simplest, most restrictive, and fastest form of coroutines. There are also coroutines that have their own stacks and coroutines that are best described as non-preemptive threads. We'd like something like that also for C++, but those are not yet quite ready (as far as I know).

As an aside, I can point out that for its first 10 years, C++ had a fast coroutine library (the task library) that was the basis for many early applications. Without the coroutines in the task library, you'd never have heard of C++. Unfortunately, the task library was not considered sufficiently user friendly, so the non-AT&T implementations didn't ship it, and it didn't make it into the standard.

* Which of these improvements in C++ 17 will have the most impact on developers
    * (parts of) Library Fundamentals TS v1
    * Parallelism TS v1
    * File System TS v1
    * Special math functions
    * hardware_*_interference_size
    * .is_always_lockfree()
    * clamp()
    * non-const .data() for string

It depends who you are and what you are doing. For me, I suspect the parallel algorithms will be the most important and having the file system library will be a nice convenience. For some, optional, any, and string_view from the Library Fundamentals will be significant. There are also many small improvements all over the standard library.

If you do serious math (e.g., physics computation, data science, or statistics) the "special math functions" (e.g., Bessel functions) are essential so it is good that they are now in the standard.

* Is this still an accurate reflection of what is planned for C++ 17?

Unfortunately, it is not. That interview reflects what I hoped for and what I considered feasible then. I presented another summary of what I was aiming at for C++17 to the committee last year. It was unlikely that we could get all of that, but I had not expected that we would get hardly any. Roll on 2020!

But note that much (maybe even most) of what I wished for is available today! Most are in TSs. You can use these features if you are willing to use beta versions.

* Isn't C++ being leveraged a lot for low-level mobile development?

Yes, that too. C++ is used in most infrastructure software, in games, in finance, and much, much more. For example see this survey. Yes, it appears that there is now well over 4 million C++ programmers.

* Will all facets of C++17 be known in July?

I hope so. We have a number of smaller proposals to decide about in Oulu, Finland, in late June 2016. For example:

- Dynamic memory allocation for over-aligned data (for better vectorization)
- Template parameter deduction for constructors (make many "make functions" redundant).
- constexpr_if (a compile-time if)

- Refining Expression Evaluation Order for Idiomatic C++ (finally, we can eliminate bugs from people accidentally relying on undefined order of evaluation)
- Default comparisons (==, !=, <, <=, >, and >=)
- Operator Dot (smart references)
- Generalizing the Range-Based For Loop (for sentinel-based and counted ranges)
- Structured bindings (simple use of multiple return types)

With a bit of luck, most will make it; but then again, we can't be sure about anything until the votes are counted. The committee strives for consensus so it doesn't take many objectors to keep a proposal out of the standard. A "no" vote counts about as much as five "yes" votes.

Should we get most of those, C++17 will become much more interesting compared to what was approved at the March 2016 meeting.

* I presume you're still directly involved in C++'s development?

Certainly, I just came back from six grueling days at the standards meeting in Jacksonville, Florida. I write and evaluate proposals, I experiment (e.g., with multimethods and FP-style pattern matching – see my publication list). It's hard work and occasionally pretty tough, but when something works out, the benefits are immense for millions of developers, and through them, billions of users of their work.

* Also, what are you up to these days? Still a professor at Texas A&M?

My "day job" is in the technology division of Morgan Stanley, a large commercial bank. I mostly deal with issues of performance and reliability; that is very much what I have always been doing. This involves a fair bit of networking and distributed systems work. There is a lot of C++ code and more being constructed every day.

This is one of the reasons for my (collaborative) work on the C++ Core Guidelines to help people write better, more modern C++. Recently, we figured out how to write completely type- and resource-safe C++ and the tools for ensuring that are on their way.

I now live in New York City where I sometimes give a course at Columbia University. I retain a connection with Texas A&M University as a University Distinguished Professor.