SC22/WG21/N4492
2015-05-15
Bjarne Stroustrup (bs@ms.com)

Thoughts about C++17

Abstract

This note was originally written to focus a discussion at the Lenexa (Kansas) committee meeting. It is a collection of thoughts aimed at stimulating a discussion, rather than a formal paper or a precise proposal. It lists suggestions for what C++17 could be, both in general and personal preferences, and expresses some opinions of what we must do to meet those goals.

Introduction

The original version of this note was written for committee members, but "escaped into the wild." Here are a few comments from the web:

- http://www.reddit.com/r/programming/comments/33us7z/what_will_c17_be_bjarne_stroustrup_on_c17_goals/
- http://forums.theregister.co.uk/forum/1/2015/04/27/c_daddy_bjarne_stroustrup_outlines_directions_for_v17/#c_2500733
- https://news.ycombinator.com/item?id=9441245

As you see, people outside the committee also have strong opinions. Those opinions can depart radically from the ones I hear in the committee and from reality. This version of the note is based on feedback from many sources, including my presentation on this topic at the Lenexa meeting.

I am often asked "What will C++17 be?" and variations of "What will C++17 do for me?" That is, "often" as in "more than twice most weeks." That embarrasses me immensely because I cannot in a simple way articulate what C++17 is supposed to become. A list of more or less related language features and standard-library components is not an acceptable answer.

I could articulate our aims for C++98, C++11, and C++14. What we produced mostly reflected those aims.

It seems to be a popular pastime to condemn C++ for being a filthy mess caused by rampant design-by-committee. This has been suggested repeatedly since before the committee was founded, but I feel the situation is now far worse. C++ is larger now (especially when we consider the standard library). That, and the variety of current proposals make that accusation credible. It really annoys me when people try to show how clever they are by presenting convoluted puzzles that I think belong in the "It hurts? Of course hitting yourself in the head with a (metaphorical) hammer hurts; so just don't do that!" category. They blame the committee. Such code examples are often used against C++. Not to help people write better software, but to scare people off learning C++. Many of the problems we address using C++ are messy, messier than many are willing to believe, and some of that messiness leaks into the language. Also, we cannot clean up old messes: doing so could break billions of lines of C++ code.

It is far easier to criticize than to build.

Robert Klarer points out: "I think the fears of 'design by committee' are way overblown. For one thing, nearly everything in the world has been designed by a committee, anyway. … Very few things that are made by humans are not the product of some kind of collaboration." We need to work as a team; there is no other way.

C++ is a great tool for many uses, especially for demanding uses involving resource constraints, large scale, and long life. We want it to be even better at that and also make it more approachable for novices and people who need to tackle less demanding jobs (but often in a hurry).

The world changes, we change, how could a language not change?

That said, we have to be careful when considering changes. C++ is huge and every addition needs to be documented and popularized. Many users rely on decades' old (and often faulty) information about C++. Some will mistake progress for instability. Every addition must be considered in the context of the usability of the whole language and of its reputation.

What is C++17?

So what is C++17 supposed to be? Here is my first cut:

- Improve support for large-scale dependable software
- Provide support for higher-level concurrency models
- Simplify core language use, especially as it relates to the STL and concurrency, and address major sources of errors.

If I wanted to, I could see the last two aims as refinements of the first.

These are relatively simple to express and a direct continuation of what we did for C++11 and C++14. In fact, they differ little for the long-term aims of C++ from its inception:

- B. Stroustrup: A History of C++: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2).
- B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007. (incl. slides and videos).
- B. Stroustrup: The Design and Evolution of C++. Addison Wesley, ISBN 0-201-54330-3.

Whatever we do, we should preserve C++'s fundamental strengths:

- A direct map to hardware (initially from C)
- Zero-overhead abstraction (initially from Simula)

Depart from these and the language is no longer C++. There are many languages with different aims. They do not provide panaceas either. By sticking to its original aims while being open to innovative ways to better approximate those ideals, C++ has become one of the most successful languages in the history of computing. To continue that, we must avoid the dual traps:

- Abandoning the past (e.g., by seriously breaking compatibility – C++ is heavily used for long-lasting systems)
- Failing to address newer challenges (e.g., by not supporting higher-level concurrency models – C++ is heavily used to address concurrency needs)

OK. That's motherhood and apple pie. Here are some details (with links to some of the relevant papers):

- Improve support for large-scale dependable software
  - ***Modules*** (to improve locality and improve compile time; http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4465.pdf and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4466.pdf)
  - ***Contracts*** (for improved specification; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4378.pdf and http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4415.pdf)
  - A ***type-safe union*** (probably functional-programming style pattern matching; something based on my Urbana presentation, which relied on the Mach7 library: Yuriy Solodkyy, Gabriel Dos Reis and Bjarne Stroustrup: Open Pattern Matching for C++. ACM GPCE'13.)
- Provide support for higher-level concurrency models
  - Basic ***networking*** (e.g. asio, http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4478.html)
  - A ***SIMD vector*** (to better utilize modern high-performance hardware; e.g., http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4454.pdf but I'd like a real vector rather than just a way of writing parallelizable loops)
  - Improved ***futures*** (e.g., http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3865.pdf)
  - ***Co-routines*** (finally, again for the first time since 1990; https://isocpp.org/files/papers/N4402.pdf, https://isocpp.org/files/papers/N4403.pdf , and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4398.pdf)
  - ***Transactional memory*** http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4302.pdf)
  - ***Parallel algorithms*** (incl. parallel versions of some of the STL; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf)
- Simplify core language use, especially as it relates to the STL and concurrency, and address major sources of errors.
  - ***Concepts*** (for better generic programming; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3701.pdf and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4361.pdf)
  - ***Concepts in the standard library*** (based on the work done on Ranges http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4263.pdf)
  - ***Ranges*** (simplifies STL use, among other things; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4382.pdf)
  - ***Default comparisons*** (to complete the support for fundamental operations; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4476.pdf)
  - ***Uniform call syntax*** (among other things: it helps concepts and STL style library use; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4474.pdf)

- ***Operator dot*** (to finally get proxies and smart references; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4477.pdf)
- **array_view** and **string_view** (better range checking, DMR wanted those: "fat pointers"; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html)
- arrays on the stack (**stack_array** anyone? But we need to find a safe way of dealing with stack overflow; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4294.pdf)
- **optional** (unless it is subsumed by pattern matching, and I think not in time for C++17, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html)

Note that I deliberately didn't segregate library components and language features. That distinction is not meaningful for users.

I assume that C++17 will ship in 2017. I do not suggest we go back to the "standard ISO" 10-year release cycle.

If your favorite feature isn't on my list, does it fit into the framework? Should the list of aims be augmented? I think three major bullet points is the limit and that the last two suggested bullets are inevitable. Can you suggest improvement to the high-level aims list? (Formulation? Substance?).

I see the support for concurrency as necessary to sustain C++'s traditional support for direct mapping to hardware. Modern hardware relies critically on concurrency and parallelism for performance and modern applications are often distributed.

I see simplifying use as necessary for coping with our larger and more complex systems. We need a better foundation (or language features and standard-library facilities) not to get lost in a mess of very low-level details. This is the C++ abstraction mechanisms being enhanced and put to good use.

If your favorite feature doesn't fit, would it fit in the C++20 time frame supporting a general aim that you can articulate? My current proposals are there because I chose to work on what I considered important and feasible in the near-to-medium term (C++17, C++20), rather than just because they are mine.

We have a long tradition adding something that's obviously useful and well tried out even if it doesn't fit the stated aims/ideals. Complex<T> comes to mind. I won't rule out doing that again, but we shouldn't overdo it. Do not let such "unfocussed ideas" divert us from the core aims or the deadline.

How to fail

C++17 is supposed to be a major release as opposed to the minor releases C++03 and C++14. Many people will be most disappointed unless we deliver something major. My guess is that two or three major features, a few medium ones, and a few minor ones would be the minimum acceptable and also feasible. I deem something "major" or "minor" based on what it does for users, rather than on their impact on the standard text. So **concept**s, **module**s, and ranges are major (and among my favorites), but default comparisons and **std::optional** are minor (though still among my favorites). However, let's try not to let a discussion about what is major and what is minor or about whether 2, 5, or 7 are the right number of features in a category distract us from the more serious tasks of improving C++.

C++ use is on the increase. We need to improve C++ to sustain the momentum, not better serve the C++ programmers, not out of fear of competition.

Adding a lot of unrelated features and library components will do much to add complexity to the language, making it scarier to both novices and "mainline programmers." It will please some experts (some of us and our friends), but unless we can articulate our aims and guiding principles, it will do little to attract programmers. You may call that "marketing", but I think of it as "design."

To be concrete, I consider the STL a model example: had Alex Stepanov proposed just a few containers or just a few algorithms, that proposal would have appeared to be simpler, but it would have had far less positive effect than the STL. In fact, the STL was transformative. I hope that concepts and a standard library based on concepts will be another model example, rather than an example of how "design by committee" and risk aversion can derail a good idea.

What I do not want to try to do:

- Turn C++ into a radically different language
- Turn parts of C++ into a much higher-level language by providing a segregated sub-language
- Have C++ compete with every other language by adding as many of their features as we can
- Incrementally modify C++ to support a whole new "paradigm"
- Hamper C++'s use for the most demanding systems programming tasks
- Increase the complexity of C++ use for the 99% for the benefit of the 1% (us and our best friends)

I think each of those would fail.

Could we agree on that? Sure, but could we stick to an agreement? I doubt it. We can relatively easily agree on high-level aims, but when the implication is that something cannot be done or that something must be postponed to meet the deadline, agreement on principles easily evaporates. There is a saying "it's deciding what to leave out that's hard."

Obviously, it is not my suggestion that we should stop working on proposals to improve C++ except for a specific list, but my fear is that unless we agree on something and prioritize, we will end up with a dozen “almost ready” major features in 2016 when the feature freeze must happen but nothing major ready to ship. A dozen minor, isolated features will not compensate because they would not significantly change the way we construct our programs. A lack of supporting facilities can do harm. For example, concepts without a concept-enabled STL could deny the benefits of concepts (better specification and better error messages) to the majority of programmers. Fortunately, the range proposal is well on its way to eliminate that potential problem.

Most people are passionate about their proposals, but we can’t accept all. In particular, there will be proposals that we all agree are improvements that we can’t ship in 2017. I too have had proposals that didn’t make into C++. In some cases, leaving them out was the right decision. In some cases, a delay led to improvements. And in some cases, it would have been better if the feature had been accepted. Let’s try not to take delays and rejection of proposals too personal. Yes, that can be hard.

To paraphrase Pete Becker: That’s the iron triangle: features, quality, and shipping date. Choose at most two. By not choosing, we would decide to choose none.

Bad (committee) habits to avoid, to avoid giving critics more ammunition:

- Make something a library because that's easier to get accepted in the committee than a language feature (even if there is good argument that what is provided is fundamental)
- Given a choice between two alternatives, choose both, add a third, and modify the first two "to please everybody who could affect the vote" (this is pure design-by- committee)
- Oppose proposals seen as competing with your favorite proposal for time/resources
- Push hard for the immediately useful (only)
- Oppose proposals not relevant to your current job, stalling an improvement that would benefit others
- Focus on the Working Paper (WP) text and choose among technical alternatives based on what fits best with the current text, rather than giving precedence to user needs
- Think that more syntax equates to safety and ease of use for the majority of programmers
- Think that “no keywords” implies simplicity and usability
- Serve the library writers and other experts while ignoring the majority of current and potential C++ programmers
- Chase “current fashion” and push for features from other popular languages without considering their impact on C++ programming styles or interactions with existing C++ facilities
- Try to make a proposal self-contained to avoid interaction with existing facilities
- Consider only solutions that are obvious or fashionable, rather than addressing root causes of programmers’ problems
- Ignore other proposals being considered
- Present “principles” as non-negotiable absolutes
- Accept nothing that is not perfect
- Consider minor improvements inherently of minor importance
- Try to do “everything”
- Add something just because we can

Try not to do any of that!

No, I did not say that would be easy. There are many more ways of failing than there are of succeeding.

We need to ship something we can be proud of and that we can articulate. We need something that’s coherent, reasonably complete at what it does, implementable (without heroics), and teachable.

Remember that every extension carries a cost. It is not just an issue of implementability and run-time cost. There is also compile-time cost, documentation cost, teaching cost, need to explore alternatives, need to explore interactions with other features, need to explore implications on programming style (both positive and negative), implications for tool building, and much more. To make decisions harder, the absence of a feature typically also carries cost (by pushing complexity into application code). We always have to make a judgment based on perceived benefits and perceived costs. This is true for standard library extensions as well as for new language features.

One way of thinking about language and standard-library features is not just to ask “will this help someone?” Most suggested features do, but “will it harm anyone” and especially “is this feature among the top 20 for helping C++ programmers?” If the answer is “I don’t know” to the first question or “no” to the second, maybe the proposal needs to be refined or dropped.

We need long-term aims, but those have a tendency to turn into “motherhood and apple pie.” We also need short-term aims or we will simply add features that address someone's current problems in the most expedient way, possibly blocking future progress or leading to duplication of facilities. That's what I consider “the developer mindset”, which is unsuitable for a long-term project with effects spanning decades. We design for decades, not just for the next release. We need to articulate our aims. The first of my suggested major aims was biased towards the longer term, the other two more towards C++17.

In the standards work we cannot ***just*** be developers. To succeed, we have to cover many roles from research, through goal setting, planning, management, and implementation, to education and popularization. And probably more.

We need to ship something major in C++17, and I think we are on track for that. I think it would be a big mistake to use the “train model” as an excuse to delay shipping or to focus only on small improvements. The implication is that we must focus on a few major issues and on smaller issues that support those. What I fear is that we try to do everything and give every proposal equal weight, so that all we get is a set of minor improvements going in all directions. Obviously, some things must be postponed to C++20. We have study groups and TSs (ISO Technical Specifications) both to get facilities ready for the standard and to have concrete artifacts that are useful before they get into the standard. Not all facilities we are working on can get into C++ and not all of the ones we like can get into C++17. I hope we are mature enough as a (large) group not to “game the system” to the detriment to the overall progress of the language (incl. the standard library).

What happens if we don’t ship something major in 2017? I suspect that implementers will just proceed to implement what they and their employers and paying customers consider important. There was a time where implementers were significantly behind in implementing the standard. This is no longer true. If implementers proceed with features “stuck in the ISO process”, we could be faced with competing feature sets, competing “implementation details” of widely implemented features, and implementers competing by offering a superset of proposed features. Nobody, not even the implementers (sometimes, especially not the implementers), would like to see such a free for all. To avoid this, the C++ standards committee must be responsive to user demand, but it must also have a focus so as to avoid ‘throwing in the kitchen sink.”

Keep simple things simple for the majority of programmers. Note that **auto** and range-**for** loops are invariably near the top of people’s list of useful C++11 features. They are also among the simplest facilities we provided.

Why am I not proposing the major libraries, frameworks, and tools that would obviously be the greatest help to real-world programmers? For example, encryption, 3-D graphics, a GUI framework, XML and Jason support, better Unicode support, comprehensive linear algebra, compression, image processing, non-shared memory distributed programming, a map-reduce framework, FIX support, protein folding, email interfaces, chat interfaces, animation frameworks, gaming engines, etc., etc. All of these exist in C++ and are used in production. Obviously, for any given programmer some such library/tool would be of far greater immediate importance than any improvement to the language or standard library. However, the standards committee does not have the resources (in time, money, and specialized skills) to take on such major tasks unless someone comes along with a specific, concrete, widely useful, well

documented, reasonably uncontroversial, and with open source implementation. Even given that and with the best will in the world, the committee may not be able to agree on a formal specification. My guess is for such major components, a community website of carefully evaluated components would stand a better chance of succeeding. For some kinds of foundation libraries boost is an answer. For others a TS might be a feasible path.

My hopes for C++17

"So, Bjarne," some of my friends say with a smile, "tell us what you ***really*** think!" indicating that I have already been too blunt and probably insulted someone. Nothing I said should insult anyone who has the best interests of the C++community at heart. In fact, nothing I say is meant to insult anyone.

So here is my top-ten list for C++17 (no order within the list):

- * ***Concepts*** (they allows us to precisely specify our generic programs and address the most vocal complaints about the quality of error messages)
- * ***Modules*** (provided they can demonstrate significant isolation from macros and a significant improvement in compile times)
- * ***Ranges*** and other ***key STL components*** using concepts (to improve error messages for mainstream users and improved the precision of the library specification "STL2")
- ***Uniform call syntax*** (to simplify the specification and use of template libraries)
- * ***Co-routines*** (should be very fast and simple)
- * ***Networking*** support (based on the asio in the TS)
- * ***Contracts*** (not necessarily used in the C++17 library specification)
- * ***SIMD vector and parallel algorithms***
- * ***Library "vocabulary types"***, such as **optional**, **variant**, **string_view**, and **array_view**
- * A "magic type" providing arrays on the stack (**stack_array**) with support for reasonable safe and convenient use.

This is my list, based on what I think feasible and would best serve the C++ community. Your list may very well be different. Note that there are proposals that I like – even proposals I'm first author of – that did not make it onto this short-list. There are problems that ideally should be addressed, but that I don't have a technical solution for, such as a standard C++ ABI. I think that there are several more proposals that could be delivered in C++17, so please don't consider this a final list. It is an attempt to focus a constructive discussion, not to close doors. Also, we will have TSs.

I consider this complete list feasible, to be delivered in the standards text in 2017, and in compilers and libraries at the same time. Most exist in some form or other today. All would impact how most of us would design and write code. I added a * to each item that made significant progress in Lenexa (and/or at the Frankfurt libraries meeting). Feel free to conclude that I'm still an optimist.

Finally, I will repeat the plea for consistency and coherence (of the language and the library, of the various new features, and of the new and older features together) that prompted me to write (and revise) this note. Our aim must be a coherent language serving articulated ideals, rather than just a bunch of semi-related features.

After C++17, C++20 will be there to "complete C++17" just as C++14 completed C++11.