C++ IS schedule

Document Number: **P1000R2** Date: 2018-07-29
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: WG21

R2: Added procedure for approving schedule exceptions, and FAQs.

IS schedule

The following is the current schedule for the C++ IS, approved by WG21 unanimous consent in Jacksonville (2018-03).

2017.2 – Toronto	First meeting of C++20
2017.3 – Albuquerque	*Try to front-load "big" language features including ones with broad library impact*
2018.1 – Jacksonville	*(incl. try to merge TSes here)*
2018.2 – Rapperswil	*EWG: Last meeting for new C++20 language proposals we haven't seen before*
2018.3 – San Diego	*EWG → LEWG: Last meeting to approve C++20 features needing library response*
	LEWG: Focus on progressing papers on how to react to new language features
2019.1 – Kona	** → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)*
	C++20 design is feature-complete
2019.2 – Cologne	CWG+LWG: Complete CD wording
	EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions
	C++20 draft wording is feature complete, **start CD ballot**
2019.3 – Belfast	CD ballot comment resolution
2020.1 – Prague	CD ballot comment resolution
	C++20 technically finalized, **start DIS ballot**

When a proposal is (or may be) "late"

In cases where we receive a proposal that may be late (comes after some deadline on this schedule, such as proposing something that may be considered a new feature request after the "feature-complete" deadline), exceptions to this schedule can be approved by strong consensus at both the design subgroup and WG21 levels.

If we receive a proposal that at least one WG21 national body expert thinks is "too late," then the following procedure applies.

In EWG/LEWG subgroups, when handling such a proposal:

- The group will first take a procedural poll on whether they have strong consensus that the proposal can be considered for the current IS cycle, where experts may vote "favor" either because they think it is not actually after a deadline (e.g., is a bug-fix, not a new feature request past the feature-complete deadline) or because they think it is worth making a past-deadline exception to the schedule. We look for strong consensus because if the subgroup itself does not have strong consensus, then the proposal is unlikely to achieve strong consensus in plenary to get the same procedural exception.
- If that poll succeeds, the group then continues with normal technical discussion about it for this IS cycle, but again requires strong consensus to approve a change. Otherwise, the group can continue with normal technical discussion about it but for some target ship vehicle that is not this IS cycle.
- In subgroups, "strong consensus" means 3:1 #favor:#against and more than half of total votes in favor (greater than the usual 2:1).

In WG21 plenary, when the subgroups bring a poll to plenary to adopt such a proposal for this cycle:

- The group will first take a procedural poll on whether they have strong consensus, in both of individual WG21 national body expert positions and national positions, that the proposal can be considered for the current IS cycle, where experts/national may vote "favor" either because they think it is not actually after a deadline (e.g., is a bug-fix, not a new feature request past the feature-complete deadline) or because they think it is worth making a past-deadline exception to the schedule.
- If that poll succeeds, the group then continues to take the normal technical adoption poll for this IS cycle, but again requires strong consensus to approve a change. Otherwise, the poll is struck (not taken).
- In WG21 plenary, "strong consensus" means 4:1 #favor:#against and more than half of total votes in favor (greater than the usual 3:1).

FAQs

(FAQ in year 20<NN-1>) There are bugs in the draft standard, should we delay C++<NN>?

Of course, and no.

Fixing bugs is the purpose of the final year, and it's why this schedule set the feature freeze deadline for C++<NN> in early 20<NN-1> (a year before), to leave a year to get a round of international comments and apply that review feedback and any other issue resolutions and bug fixes.

If we had just another meeting or two, we could add <feature> which is almost ready, so should we delay C++<NN>?

Of course, and no.

Just wait a couple more meetings and C++<NN+3> will be open for business and <feature> can be the first thing voted into the C++<NN+3> working draft. For example, that's what we did with concepts; it was not quite ready to be rushed from its TS straight into C++17, so the core feature was voted into draft C++20 at the first meeting of C++20 (Toronto), leaving plenty of time to refine and adopt the remaining controversial part of the TS that needed a little more bake time (the non-"template" syntax) which was adopted the following year (San Diego). Now we have the whole thing.

This feels overly strict. Why do we ship IS releases at fixed time intervals (three years)?

Because it's one of only two basic project management options to release the C++ IS, and experience has demonstrated that it's better than the other option.

What are the two project management options to release the C++ IS?

I'm glad you asked.

There are two basic release target choices: Pick the features, or pick the release time, and whichever you pick means relinquishing control over determining the other. It is not possible to control both at once. They can be summarized as follows:

If we choose to control this	**We give up control of this**	**Can we work on "big" many-year features?**	**When do we merge features into the IS working draft?**	**What do we do if we find problems with a merged feature?**
"What": The features we ship	"When": The release time	Yes, in proposal papers and the IS working draft	Typically earlier, to get more integration testing ⇒ lowers average working draft stability	Delay the standard
"When": The release time	"What": The features we ship	Yes, in proposal papers and TS "feature branches"	Typically later, when the feature is more baked ⇒ increases average working draft stability	Pull the feature out, can merge it again when it's ready on the next IS "train" to leave the station

Elaborating:

(1) "What": Pick the features, and ship when they're ready; you don't get to pick the release time. If you discover that a feature in the draft standard needs more bake time, you delay the world until it's ready. You work on big long-pole features that require multiple years of development by making a release big enough to cover the necessary development time, then try to stop working on new features entirely while stabilizing the release (a big join point).

This was the model for C++98 (originally expected to ship around 1994; Bjarne originally said if it didn't ship by about then it would be a failure) and C++11 (called 0x because x was expected to be around 7).

This model "left the patient open" for indeterminate periods and led to delayed integration testing and release. It led to great uncertainty in the marketplace wondering when the committee would ship the next standard, or even if it would ever ship (yes, among the community, the implementers, and even within the committee, some had serious doubts in both 1996 and 2009 whether we would ever ship the respective release). During this time, most compilers were routinely several years behind implementing the standard, because who knew how many more incompatible changes the committee would make while tinkering with the release, or when it would even ship? This led to wide variation and fragmentation in the C++ support of compilers available to the community.

Why did we do that? Because we were inexperienced and optimistic: (1) is the road paved with the best of intentions. In 1994/5/6, and again in 2007/8/9, we really believed that if we just slipped another meeting or three we'd be done, and each time we ended up slipping up to four years. We learned the hard way that there's really no such thing as slipping by one year, or even two.

Fortunately, this has changed, with option (2)…

(2) "When": Pick the release time, and ship what features are ready; you don't get to pick the feature set. If you discover that a feature in the draft standard needs more bake time, you yank it and ship what's ready. You can still work on big long-pole features that require multiple releases' worth of development time, by simply doing that work off to the side in "branches," and merging them to the trunk/master IS when they're ready, and you are constantly working on features because every feature's development is nicely decoupled from an actual ship vehicle until it's ready (no big join point).

This has been the model since 2012, and we don't want to go back. It "closes the patient" regularly and leads to sustaining higher quality by forcing regular integration and not merging work into the IS draft until it has reached a decent level of stability, usually in a feature branch. It also creates a predictable ship cycle for the industry to rely on and plan for. During this time, compilers have been shipping conforming implementations sooner and sooner after each standard (which had never happened before), and in 2020 we expect multiple fully conforming implementations the same year the standard is published (which has never happened before). This is nothing but goodness for the whole market – implementers, users, educators, everyone.

Also, note that since we went to (2), we've also been shipping more work (as measured by big/medium/small feature counts) at higher quality (as measured by a sharp reduction in defect reports and comments on review drafts of each standard), while shipping whatever is ready (and if anything isn't, deferring just that).

How serious are we about (2)? What if a major feature by a prominent committee member was "almost ready"… we'd be tempted to wait then, wouldn't we?

Very serious, and no.

We have historical data: In Jacksonville 2016, at the feature cutoff for C++17, Bjarne Stroustrup made a plea in plenary for including concepts in C++17. When it failed to get consensus, Stroustrup was directly asked if he would like to delay C++17 for a year to get concepts in. Stroustrup said No without any hesitation or hedging, and added that C++17 without concepts was more important than a C++18 or possibly C++19 with concepts, even though Stroustrup had worked on concepts for about 15 years. The real choice then was between: (2) shipping C++17 without concepts and then C++20 with concepts

(which we did), or (1) renaming C++17 to C++20 which is equivalent to (2) except for skipping C++17 and not shipping what was already ready for C++17.

Why not every { one, two, four } years?

We find three years to be a good balance, and two years is the effective minimum in the ISO process.

What about something between (1) and (2), say do basically (2) but with "a little" schedule flexibility to take "a little" extra time when we feel we need it for a feature?

No, because that would be (1).

The 'mythical small slip' was explained by Fred Brooks in *The Mythical Man-Month*, with the conclusion: "Take no small slips." For a moment, imagine we did slip C++<NN>. The reality, regardless of any valiant efforts to live in denial of it, is that we would be switching from (2) back to (1). If we had decided to delay C++<NN> for more fit-and-finish, we will delay the standard by at least two years. There is no such thing as a one-meeting or three-meeting slip, because during this time other people will continue to (rightly) say "well my feature only needs one more meeting too, since we're slipping a meeting let's add that too." And once we slip at least two years, we're saying that C++<NN> becomes C++<NN+2> or more likely C++<NN+3>… but we're already going to ship C++<NN+3>! — So we'd still be shipping C++<NN+3> on either plan, and the only difference is that we're *not* shipping C++<NN> in the meantime with the large amount of work that's stable and ready to go, and making the world wait three more years for it. And gratuitously so, because the delay will not benefit those baked features, which is most or all of them.

So the suggestion to "slip C++<NN>" amounts to "rename C++<NN> to C++<NN+2> or C++<NN+3>," and the simple answer is "yes, we're going to have C++<NN+3> too, but in addition to C++<NN> and not instead of it."

But feature X is broken / needs more bake time than we have bugfix time left in C++20!

No problem! We can just pull it, as we did with draft C++11 concepts and with draft C++20 contracts.

Under our current plan (2), someone needs to write the paper aimed at EWG or LEWG (as appropriate) that shows the problem and/or the future doors we're closing, and proposes removing it from the IS working draft. Those groups will consider it, and if they decide the feature is broken (and plenary agrees), that's fine, the feature will be delayed to C++next.

But under plan (1), we would be delaying, not only that feature, but *the entire feature set* of C++<NN> to C++<NN+3>. That would be… excessive.

Does (2) mean "major/minor" releases?

No. We said that at first, before we understood that (2) really simply means you don't get to pick the feature set, not even at a "major/minor" granularity.

Model (2) simply means "ship what's ready." That leads to releases that are:

- similarly sized (aka regular medium-sized) for "smaller" features because those tend to take shorter lead times (say, < 3 years each) and so generally we see similar numbers completed per release; and

- variable sized (aka lumpy) for “bigger” features that take longer lead times (say, > 3 years each) and each IS release gets whichever of those mature to the point of becoming ready to merge during that IS’s time window, so sometimes there will be more than others.

So C++14 and C++17 were relatively small, because a lot of the standardization work during that time was taking place in long-pole features that lived in proposal papers (e.g., contracts) and TS “feature branches” (e.g., concepts).

C++20 is a big release, therefore didn’t we cram a lot into a three-year cycle for C++20?

No, see “lumpy” above.

C++20 is big, not because we did more work in those three years, but because many long-pole items (including at least two that have been worked on in their current form since 2012, off to the side as P-proposals and TS “branches”) happened to mature and get consensus to merge into the IS draft in the same release cycle.

It has pretty much always been true that major features take many years. The main difference between plan (1) for C++98 and C++11 and plan (2) now is: In C++98 and C++11 we held the standard until they were all ready, now we still ship those big ones once they’re ready but we also ship other things that are ready in the meantime instead of going totally dark.

C++20 is the same three-year cycle as C++14 and C++17; it’s not that we did more in these three years than in the previous two three-year cycles, it’s just that more long-pole major features that we invested in during the C++14 and C++17 cycles became ready to merge for C++20, so we’re actually reaping the work we started investing in during other cycles. And if any really are unready, fine, we can just pull them again and let them bake more for C++23. If there is, we need that to be explained in a paper that proposes pulling it, and why, for it to be actionable.

I think the right way to think about it is that C++14+17+20 taken as a whole is our third nine-year cycle (2011-2020), after C++98 (1989-1998) and C++11 (2002-2011), but because we were on plan (2) we *also* shipped the parts that were ready at the three- and six-year points.

Isn’t it better to catch bugs while the product is in development, vs. after it has been released to customers?

Of course.

But if we’re talking about that as a reason to delay the C++ standard, the question implies two false premises: (a) it assumes the features haven’t been released and used before the standard ships (many already have production usage experience); and (b) it assumes all the features can be used together before the standard ships (they can’t).

Elaborating:

Re (a): Most major C++20 features have been implemented in essentially their current draft standard form in at least one shipping compiler, and in most cases actually used in production code (i.e., has already been released to customers who are very happy with it). For example, coroutines (adopted only five months ago as of this writing) has been used in production in MSVC for two years and in Clang for at least a year with very happy customers at scale (e.g., Azure, Facebook).

Re (b): The reality is that we aren't going to find many feature interaction problems until users are using them in production, which generally means until after the standard ships, because implementers will generally wait until the standard ships to implement most things. That's why when we show any uncertainty about when we ship, what generally happens is that implementations wait – oh, they'll implement a few things, but they will hit Pause on implementing the whole thing until they know we're ready to set it in stone. Ask <favorite compiler> team what happened when they implemented <major feature before it was in a published standard>. In a number of cases, they had to implement it more than once, and break customers more than once. So it's reasonable for implementers to wait for the committee to ship.

Finally, don't forget the feature interaction problem. In addition to shipping when we are ready, we need time after we ship to find and fix problems with interactions among features and add support for such interactions that we on typically cannot know before widespread use of new features. No matter how long we delay the standard, there will be interactions we can't discover until much later. The key is to manage that risk with design flexibility to adjust the features in a compatible way, not to wait until all risk is done.

The standard is never perfect… don't we ship mistakes?

Yes.

If we see a feature that's not ready, yes we should pull it.

If we see a feature that could be better, but we know that the change can be done in a backward-compatible way, that's not a reason to not ship it now; it can be fixed compatibly in C++next.

We do intentionally ship features we plan to further improve, as long as we have good confidence we can do so in a backward-compatible way.

But shouldn't we aim to minimize shipping mistakes?

Yes. We do aim for that.

However, we don't aim to eliminate all risk. There is also a risk and (opportunity) cost to not shipping something we think is ready. So far, we've been right most of the time.

Are we sure that our quality now is better than when were on plan (1)?

Yes.

By objective metrics, notably national body comment volume and defect reports, C++14 and C++17 have been our most stable releases ever, each about 3-4× better on those metrics than C++98 or C++11. And the reason is because we ship regularly, and put big items into TS branches first (including full wording on how they integrate with the trunk standard) and merge them later when we know they're more baked.

In fact, since 2012 the core standard has *always* been maintained in a near-ship-ready state (so that even our working drafts are at least as high quality as the shipped C++98 and C++11 standards). That never happened before 2012, where we would often keep the patient open with long issues lists and organs lying around nearby that we meant to put back soon; now we know we can meet the schedule at high quality because we always stay close to a ship-ready state. If we wanted to, we could have shipped

the CD even without the work of the Cologne meeting and still been much higher quality than C++98's or C++11's CDs (and, probably, higher quality than their published standards). Given that C++98 and C++11 were successful, recognizing that we're now at strictly higher quality than that all the time means we're in a pretty good place.

C++98 and C++11 each took about 9 years and were pretty good products …

Yes: 1989-1998, and 2002-2011.

… and C++14 and C++17 were minor releases, and C++20 is major?

Again, I think the right comparable is C++14+17+20 taken as a whole: That is our third nine-year cycle, but because we were on plan (2) we *also* shipped the parts that were ready at the three- and six-year points.

Does (2) allow making feature-based targets like P0592 for C++next?

Sure! As long as it doesn't contain words like "must include these features," because that would be (1). (The R2 revision of paper P0592 is expected to make this correction.)

Aiming for a specific set of features, and giving those ones priority over others, is fine – then it's a prioritization question. We'll still take only what's ready, but we can definitely be more intentional about prioritizing what to work on first so it has the best chance of being ready as soon as possible.