C++ IS schedule

Document Number: **P1000R4** Date: 2020-02-11
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: WG21

R4: Updated table for C++23 (by adding 3 years to each row).

IS schedule

The following is the schedule for the C++23 IS.

2020.2 – Varna	First meeting of C++23
2020.3 – New York 2021.1 – Kona	*Try to front-load "big" language features including ones with broad library impact*
2021.2 – Montreal	*(incl. try to merge TSes here)*
2021.3 – tbd	*EWG: Last meeting for new C++23 language proposals we haven't seen before* *EWG → LEWG: Last meeting to approve C++23 features needing library response* *LEWG: Focus on progressing papers on how to react to new language features*
2022.1 – Portland	** → CWG,LWG: Last meeting to send proposals to wording review (incl. TS merges)* C++23 design is feature-complete
2022.2 – tbd	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++26 features + CWG/LWG design clarification questions C++23 draft wording is feature complete, **start CD ballot**
2022.3 – tbd	CD ballot comment resolution
2023.1 – tbd	CD ballot comment resolution C++23 technically finalized, **start DIS ballot**

Approving schedule exceptions

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

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 Stroustrup originally said that 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).

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.

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

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.

But if <feature championed by a prominent committee member> is "almost ready," we'd be tempted to wait, wouldn't we?

No. For example, at Jacksonville 2016 (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; C++17 without concepts was more important than a C++18 or possibly C++19 with concepts.

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." Slipping means delaying the standard for at least two years, and we're already going to ship again in three years anyway, so the effect of a "slip" is actually to "skip" a release.

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.

In the second case, sometimes we do ship a release with many new features, and then it's natural for the following release to "complete" those features and fill known holes, including adding minor parts that were intentionally deferred because they could be added later.

What if we find out a feature in the draft standard needs more bake time?

If we see a feature is not ready yet, we pull it back out to let it bake more, as we did for concepts in C++11 and contracts in C++20. It can finish baking and ship in C++next.

If we see a feature that could be better, but we know that the change can be done in a backward-compatible way, we can still ship it now. It can be completed compatibly in C++next.

We aim to minimize mistakes, but we 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.

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.