Doc. no.: P2000R5
Date: 2026-02-23
Programming Language C++
Audience: All WG21
Reply to: Daveed Vandevoorde
(daveed@vandevoorde.com)
Author: Directions Group

# DIRECTION FOR ISO C++

*J. Garland, P. McKenney, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong*

Revision History

- R5: **this paper (2026-02-23 published):**
  - New text is red.
  - Moved short/medium term to a new paper P5000 setting priorities for the next standard (C++29)
  - added pointer to P3970r0: Profiles and Safety: a call to action
  - added pointer to P4024r0:Guidance on Converging on unified proposals
  - added pointer to P4023r0: Strategic Direction for AI in C++: Governance, and Ecosystem
- R4: 2022-10-15 published
  - Added new Section 5.1 Towards a Safer C++
  - Updated Section 7.6 Balance of concerns
- R3: 2021-12-21 published
  - Impact of pandemic on C++23 and C++26
  - Safety and Security
  - C and C++ compatibility
- R2: online:
  - updated with suggestions from Dan Raviv
  - our view on ISO C++ in 2020 on balance of engineering choices
  - our view on ISO C++ online and F2F
- R1: Post-Prague:
  - rephrased and improved Section 6 on C++23
  - updated authors list in mailing to include all authors
  - Highlight in red new or significantly changed sections since last revision (P2000r0)
- R0: Feb 2020 (Pre-Prague): P2000R0 is a continuation of the P0939 series of papers.
  - Updated Section 7.x on Rationale vs Design vs Wording

**Table of Content**

## 1 Abstract

This is intended as a document updated as times go by and new facts and opinions emerge. It tries to articulate our aims for C++.

This document differs from most papers by aiming for a global view and placing individual features in context, trying to avoid delving into technical details of individual proposals except where those “details” could affect the language as a whole or its tool environments.

To allow for more frequent updates regarding specific standard releases (e.g., C++29, C++32), concrete priorities and medium-term goals have been moved to a separate "Short-Term Direction" paper (P5000). This document (P2000) now focuses on the long-term philosophy and operational principles of the committee.

Furthermore, to highlight specific or new directives and prevent readers from having to repeatedly re-read this extensive document to find targeted updates, the Direction Group is now publishing focused directives as standalone papers. Examples include **P3970r0**: Profiles and Safety: a call to action, **P4024R0** (Guidance on Converging on unified proposals) and **P4023R0** (Strategic Directions for AI in C++: Governance and Ecosystem). By extracting these high-priority topics into their own documents, we aim to make new guidelines more discoverable, actionable, not repeated with inconsistencies, and easier to

reference without getting lost in the broader philosophy. Pointers to these standalone papers are embedded in the relevant sections below as we move forward from R5. These standalone papers are intended to carry the same weight as P5000.

Suggested improvements are welcome.

## 2 History

The Direction Group (DG) was created in response to calls for more thought about the direction of C++'s evolution (language and standard library) and comments about our self-established processes in WG21. The specific "call to action" was Operating principles for C++ [Winkel,2017] by several Heads of National Delegations.

We see C++ in danger of losing coherency due to proposals based on differing and sometimes mutually contradictory design philosophies and differing stylistic tastes. For that reason, we recommend that you (re)read [Winkel,2017] before proposing a new feature (language or library).

[Winkel,2017] quoted heavily from D&E. However, during the presentation of C++ Stability, Velocity, and Deployment Plans [Winters,2017] at the June 2017 Toronto meeting to an almost complete WG21, Titus Winters asked for a show of hands of who had read "The Design and Evolution of C++" [Stroustrup,1994]. Only about a quarter of the hands went up. Assuming that another quarter was shy or distracted, that indicates that half of the committee have never read the articulation of key design principles and decisions for C++. Part of the reason for that Is that copies of D&E can be hard to find. Consequently, we offered every member of the committee a free copy and ensured that Addison Wesley made D&E available again (in paper and electronic versions).

We strongly recommend that someone who wants to push or oppose a proposal read [Stroustrup,1994] and its related HOPL papers ([Stroustrup,1993], [Stroustrup,2007], and [Stroustrup2020]). Our impression is that many members of WG21 have an inaccurate view of how and why C++ succeeded and some are pushing popular approaches that in the past have led to failure (e.g., by languages following those approaches and losing out to C++). Obviously aims, techniques, and ideals change over decades, progress happens, but it is dangerous to operate in ignorance of past successes and failures.

## 3 The Direction Group

The direction group's (DG) charter is [Sutter,2018]:

> Direction group. The direction group is a small by-invitation group of experienced participants who are asked to recommend priorities for WG21. Their charter includes setting forth a group opinion on:
>
> - Evolution direction (language and library): This includes both language and library topics, and includes both proposals in hand and proposals we do not have but should solicit. The direction group maintains a list of the proposals it considers the most important for the next version of C++ or to otherwise make progress such as in

a TS, and the design group chairs use that list to prioritize work at meetings. Typically, work on other topics will occur after there's nothing further left to do at this meeting to advance the listed items.

- Providing an opinion on any specific proposal: This includes whether the proposal should be pursued or not, and if pursued any changes that should be considered. Design group participants are strongly encouraged to give weight to an opinion that the direction group feels strongly enough about to suggest.

We are concerned with both the long-term and the shorter-term direction and handling of proposals, especially where they may intersect multiple WGs. When looking at direction and at individual proposals, we try to consider the interests of the larger C++ community, rather than just the narrower interests of WG21 members.

The direction group mailing list is direction@lists.isocpp.org. Opinions welcome.

We operate similarly to a Board of Directors, with an annually-rotating chairman. Bjarne lost a random number draw (written by Howard) and was therefore our chair for the first year:

- Michael Wong (2020, 2025)
- Daveed Vandevoorde (2021, 2026)
- Roger Orr (2022, 2027)
- Bjarne Stroustrup (2018, 2023, 2028)
- Jeff Garland (2029)
- Paul McKenney (2030)

Former members:

- Beman Dawes. It is with deep sadness that we learned of Beman's passing in early 2020. We wish to remember him not only for his contribution within this group, but also for his invaluable influence on C++ and its community (e.g., through the founding of Boost, the shepherding of standard C++ filesystem support , and his crucial early advocation of the STL), but most of all for being an extraordinary friend and human being. Just about everyone using C++ today has reason to be grateful to Beman. He is sorely missed.
- Howard Hinnant. Howard retired from the Direction Group at the end of 2024.

We only speak as "The Direction Group" when we are in unanimous agreement on a topic.

## 4 Long-term Aims (decades)

Assuming we succeed, many of us will be writing and maintaining C++ in 10 years' time and 20 years' time. What kind of aims would we think reasonable for such a time scale? Obviously, long-term aims cannot change every year.

As a community, we want lots of things for C++, but

- Some we can’t get because they are impossible.
- Some we can’t get because they are incompatible with something else we have or want to have.
- Some we don’t yet know how to do.
- Some we don’t have the resources to do.
- Some we don’t agree would be good for C++.
- Some we don’t agree would be worthwhile for C++.
- Some are not ready for standardization but may be in the future, or may never be ready.

We understand that we can’t have everything we want and that some of the things we want take a lot of time and work. Given that, we need to clarify our current long-term goals.

No language can be everything for everybody. Trying to achieve that has killed many efforts. We have to decide what C++ is supposed to be and to become.

What is C++?

- C++ is a language for defining and using light-weight abstractions
- C++ supports building resource-constrained applications and software infrastructure
- C++ supports large-scale software development

How do we want C++ to develop?

- Improve support for large-scale dependable software
- Improve support for high-level concurrency models
- Simplify language use
- Address major sources of dissatisfaction
- Address major sources of errors

We fundamentally need:

- *Stability*: Useful code “lives” for decades
- *Evolution*: The world changes and C++ must change to face new challenges.

There is an inherent tension here. We should be very reluctant to break compatibility. People always want a simpler language, a language without complicating “legacy features”, and get seriously angry if we break code that they depend on.

C++ is complicated, too complicated, yet we cannot remove significant facilities and changing them is very hard. Changing parts deemed insignificant can be risky (we need better analysis tools [Winter,2016]) and the potential gains would be insignificant. However, we badly need to simplify use of C++ and that leaves us with three alternatives:

- Provide simpler alternatives for simple uses
- Provide simplifying generalizations
- Provide alternatives to complicated and/or error-prone features

Often, a significant improvement involves a combination of those three.

Calling C++ a "multi-paradigm language" is no excuse for adding support for incompatible programming styles. If features don't smoothly interoperate, we get *de facto* dialects. Today, some of the most powerful design techniques combine aspects of traditional object-oriented programming, aspects of generic programming, aspects of functional programming, and some traditional imperative techniques. Such combinations, rather than theoretical purity, are the ideal.

- Provide features that are coherent in style (syntax and semantics) and style of use

This applies to libraries, to language features, and to combinations of the two.

Technically, C++ rests on two pillars:

- A direct map to hardware (initially from C)
- Zero-overhead abstraction in production code (initially from Simula, where it wasn't zero-overhead)

Depart from those and the language is no longer C++. We should not depart from these principles and fall into the trap of:

- Abandoning the past which can seriously jeopardize compatibility. C++ is and will continue to be heavily used in long-lasting systems.
- Not addressing new challenges such as higher-level concurrency models. Should we fail, developers will need to switch to some other framework to gain the best performance.

Over the long term, we must strengthen those two pillars:

- Better support for modern hardware (e.g., concurrency, GPUs, FPGAs, NUMA architectures, distributed systems, new memory systems)
- More expressive, simpler, and safer abstraction mechanisms (without added overhead)

Many useful abstractions should find their way into the standard library:

- In principle, the C++ standard library can be implemented in C++ plus a few "intrinsic operations" for accessing low level machine facilities.

This is a variant of the "Don't leave room for a language below C++ (except assembler)" rule of thumb from D&E. Where we depart from this principle, we get long-term problems because the semantics easily drift apart from the rest of the language.

C++ relies critically on static type safety for expressiveness, performance, and safety. The ideal is

- Complete type-safety and resource-safety (no memory corruption and no leaks)

This is achievable without added overhead, in particular without adding a garbage collector, and without restricting expressibility (see A brief introduction to C++'s model for type- and resource-safety [Stroustrup,2015b] and C++ Core Guidelines [CG,2015]). We don't consider a program littered with casts type-safe and we recognize that the ideal of complete type-safety and complete resource-safety cannot be achieved while accepting every legal C++ program (as we must for compatibility), so external tools will be needed (e.g., static analysis of lifetimes), but improved language and library support can help a lot by saving programmers from having to use verbose or error-prone features.

C++ is now used in more domains than ever. We do not specifically recommend any specific domains, as every domain is important to somebody in the larger C++ community – even if their domain isn't well-represented in the committee. But we do intend to broaden our support for domains that are well-represented by the C++ community, even non-traditional ones, as a high-level aim. However here are some areas of general concerns – often cutting across application domains – that we should not ignore:

- *Safety and security*: reduce vulnerability to intrusion and the ability to exploit intrusions (e.g., prevent unsafe input and eliminate type violations from misuse of free store). Applications include autonomous vehicles, medical, avionics safety, critical reliability systems, etc.
    a. Safety and security remain the paramount priorities for the evolution of C++. The Direction Group has officially endorsed the 'Profiles' framework as the path forward. For the DG's explicit call to action—detailing the required initial profiles (Initialization, Ranges, Resources) and our request for industry funding and implementation—please refer to the standalone directive:**P3970r0: Profiles and Safety: a call to action**.
- *Simplification*: make simple things simple to do (and thereby make C++ easier to learn).
- *Interoperability*: improve interoperability with important languages and systems (e.g., Python bindings). Modern C++ facilitates that with standard-layout types, variadic templates, lambdas, and more.
- *Support for demanding applications*: offer domain-specific support for important application areas, such as medical, finance, automotive, and games (e.g., key libraries, such as JSON, flat containers, and pool and stack allocators).
- *All system types and sizes from the small to the very large such as for example Edge, Microcontroller, Embedded, Desktop, Data Centers and Cloud Computing:*
    - *Embedded systems*: make C++ easier to use and more effective for large and small embedded systems, (that does not mean just simplified C-like C++; e.g., see [Quora1] [Quora2] [Saks,2016] [Stroustrup,2018]). In particular, embedded systems programming could be supported by more specific libraries, emphasizing compactness of data and predictability of operations (e.g., "no free store allocation").
    - Data Centers and Cloud Service Providers: makes C++ better for very large scale computing, including HPC by providing distributed, network and data placement

facilities for new forms of memory and compute balance, as well as improvement to workloads for Simulation, Big Data, and Machine Learning. It also supports high compute density, virtualization, and smart resource management.

    - To ensure C++ remains the dominant infrastructure language for these ML workloads, we must encourage the ecosystem to improve AI training data and tooling semantics. For our specific long-term strategy regarding AI, see **P4023R0: Strategic Direction for AI in C++ and SG19 strategic directions**.

- Alternatives for error-prone and unsafe facilities (like **std::variant** as an alternative to unions or pattern matching as an alternative for **std::variant**).

This is not an exhaustive list, but demonstrates high-level aims that bracket our priorities. The medium-term aims will bracket our priorities further with specific proposals that are currently in flight. Many of these goals cannot be met through changes in the standard alone. Some, such as bindings and avoidance of error-prone facilities, require improvements beyond the standard, but the standard can make such improvements easier.

We would like to see the software development tools used for C++ (such as compilers, static analyzers, refactoring tools) significantly improved. Most of this is beyond the scope of WG21, but we should try hard not to make things more difficult, e.g. by significantly increasing compile times or the cost of tool building by adding significant complexity barriers (e.g., by encouraging coding styles increasing the use of macros or brittle SFINAE).

To maximize the likelihood of delivering significant improvements,

- We discourage isolated “cute” proposals.

There is a limit to what the community can absorb in a given period of time and while a large number of new features can excite C++ enthusiasts, their addition inevitably leads to an impression of instability among many developers and among decision makers observing the evolution of C++ from a distance. Also, work on small isolated features distracts from the work on fundamental improvements (e.g., to the type system or the foundations of concurrency support), diverts resources, may complicate further improvements, and requires additions to references and teaching materials. In the years leading up to the 1998 standard, the committee members often reminded themselves of the story of the Vasa, the beautiful 17th century battleship that sank on its maiden voyage because of (among other things) insufficient work on its foundation and excessive late additions [Stroustrup,2018c].

## 5 Process Issues

We are “a bunch of volunteers.”

- Most are enthusiasts for some aspect of the language or other.
- Few have a global view (geographically, C++ community, C++ usage, C++ Language and standard library).
- Most have a strong interest in only some subset of use, language, or library.

- Most are deeply engaged with a specific form of C++ as part of their daily work.
- Our levels and kinds of relevant computer-science, design, and programming language education vary dramatically.
- Many are clever people attracted to clever solutions, so that the complexity of any software artifact tends to expand to just beyond our ability to maintain it.
- Some are devoted to ideas of perfection.
- Many are tool builders.
- Some have full control over their source code whereas others critically rely on code outside their direct control (open-source and/or commercial).
- Some operate in an environment with strong management control, others in environments without a formal management structure (and everything in-between).

This implies that we can't rely on a common vocabulary, a core set of shared values, a common set of basic ideals, or a common understanding of what's a problem. Consequently, we must spend more effort on

- articulating rationales for proposals.
- facilities for the "average programmer," who is seriously underrepresented on the committee.
- facilities aimed more at application builders than at builders of foundational libraries.

Please pay special attention to that last point. We feel that C++'s utility and reputation suffer badly from the committee lacking attention to improvements for relative novices and developers with relatively mundane requirements. Remember:

- Most C++ programmers are not like the members of the committee

We, as a committee, have no mechanism of reward (except accepting someone's proposal) or punishment (except delaying or rejecting someone's proposal). To get something accepted, we need consensus (defined as a large majority, but not necessarily unanimity). This has implications on what we can do and how we do it.

- Nothing gets done unless someone cares enough to do it
- A small vocal minority can stop any proposal at any stage of the process.

Currently, C++ is probably as popular as it has ever been and there are definitely more active members of the standards committees than ever before. One effect of this enthusiasm for C++ is to inundate us with a flood of proposals. The sheer volume of proposals leads to fewer proposals getting through the processes to get accepted. Some proposals become warped from the need to gain support in an environment where time to think and present ideas is limited. Many of these proposals come from people who are unacquainted with standardization. We understand that some new members find it hard to accept that

- Progress is less rapid than for their corporate and open-source projects.

- Concerns that are essential to them are not given priority by the committee.
- Committee members don't all understand or accept other members' experience and design principles.
- Standardizing for decades differs dramatically from shipping the next release with the fewest number of bugs.
- There are millions of programmers who use C++ in ways that differ from what they consider normal and reasonable.
- "No bugs" does not imply that a proposal is good.
- "There are bugs" does not imply that a proposal is bad (there is a saying among mathematicians that the way to recognize an important result is by the number of errors in its initial proof – it is relatively simple to provide a flawless proof for a trivial result).
- "Good enough" isn't always good enough because a second look at the problem might come up with a better, more general, or better integrated solution. We have to consider the long term (decades).
- 90% or more of the work of getting a proposal into the standard is ensuring that it fits smoothly with other facilities (language and standard library).
- Stability/compatibility is an essential feature. For example, any proposal introducing a silent semantic change is likely to lead to significant objection, so providing good technical justification would be recommended for any such proposal.

And still some improvements are urgent. However, in the context of the committee "urgent" still implies years of work/delay. Novices are not the only ones who are impatient, but we must try to channel our energies into constructive activities. We encourage members (old and new)

- to get acquainted with C++'s history and design rules
- to accept or contribute to our long-term aims for C++

The aim of most members, new or "vintage" is to improve C++ by having their favorite proposal accepted. However, the sheer number of proposals (constantly on the order of a hundred) and the number of papers (running at more than one hundred for each meeting), implies that not even every ***good*** proposal can be accepted. We encourage everyone to ask themselves

- Is my proposal among the top-20 in terms of long-term benefit to the C++ community as a whole?
- Is my proposal essential for some important application domain?

Push only very gently if the answers are "no," "sort of," or "I don't know" and redirect effort to proposals for which the answer is "Yes!" We encourage the WG chairs to

- Focus on the major high-level and intermediate-level goals
- Articulate the goals as they relate to their WG

- Prioritize proposals based on those goals
- Discourage proposals not in that scope
- Discourage proposals from being re-submitted with only minor changes after rejection (especially if the revised proposal does not include new insights into the problem to be solved).
- Encourage proposers to discuss potential disadvantages and costs of their proposals as well as their benefits. Every new feature comes with a significant cost.

All proposals consume the (limited) committee time and WG21 members should be considering the best *overall* outcome for the future of the language. Hence while small proposals to clean up non-trivial defects are welcome, discussion about these may have lower priority. If such a small proposal proves to be controversial it is probably better to withdraw, or defer, it to avoid preventing progress on more substantive items.

We are a set of interrelated committees currently with more than 500 members present at a meeting and more active via the Web. Thus some “design by committee”, or rather “design by committees,” is unavoidable. We need to consciously and systematically try to minimize those effects by building a shared sense of direction.

- We have no shared aims, no shared taste.

This is a major problem, possibly the most dangerous problem we face as a committee. For C++ to succeed, we must overcome that. For starters, we – as individuals, as SGs, as WGs, and as the committee as a whole – must devote more time to develop and articulate common understanding. In particular for every proposal being considered we should do more to emphasize the motivation and explain how it fits in the language, standard library, and common uses that we anticipate.

- The alternative is a dysfunctional committee producing an incoherent language.

We need to be more explicit about

- What general problems we are trying to address
- How a particular proposal serves those articulated aims

It would also be a great improvement if members didn’t spend all of their time in a single WG. Doing that leads to lack of insight and of trust.

- Try to spend at least one day each meeting in a WG that isn’t “your own”
- Try to read several papers from each mailing that is not aimed at “your WG”

Note that if “your” proposal progresses, you’ll have to shepherd it through different WGs, so it is good to have some understanding of how they operate.

In addition, the committee as a whole should try to avoid that

- a single organization or cohesive group gains control of a WG.

WGs must be open to inputs from many sources.

As specified in the ISO and NB charters

- the operations of the committee must be transparent.

It is only natural that proposals are first developed and discussed among a small group of friends or colleagues. However, once a proposal reaches a larger group, discussions should be made public to all members. Furthermore, proposals should be made available as complete papers in plenty of time for the members to consider them before meetings. It is not acceptable to present only an incomplete summary a week or two before a meeting and expect a serious discussion, let alone a binding vote. Similarly, it is unreasonable to expect members to decide on a large detailed proposal with only a few days' warning; remember that a mailing before a meeting consists of about a hundred papers, including many long ones.

To maximize the likelihood of delivering significant improvements and to increase the predictability of our processes, we discourage

- "Change the World" papers for proposals already in flight

We change the language and standard library by gradually building on previous work or by providing a better alternative to an existing feature. Proposals to completely change direction for a proposal already in process should be treated with suspicion and subjected to at least as much scrutiny as the proposal already in flight has been. Looking at new proposals is more exciting than working out the obscure details of an old proposal, but eventually all successful proposals go through the "polish the last details" stage, so we can't escape that through new and shiny proposals, just postpone the pain. A proposal for a radical change to a proposal already "in flight'' (i.e., in its second or later discussion) should not be allowed to delay the latter unless it comes with a paper with a detailed discussion of design, use, and implementation. When chairs become aware of people ignoring a paper in the hopes that it will fail, they might publicly challenge those people in order to bring their objections to bear earlier in the process. Should these people refuse to state objections early, any late-in-process objections may be easier to ignore. But no matter what rules we put in place, we can count on our smart, motivated, and creative committee members to game them.

## 5.1 Trust

At the start of the formal standardization efforts (ANSI and ISO), P.J. Plauger and Tom Plum emphasized the need for trust in the process. The desired degree of trust is currently missing, and the consequences are dire.

And the growth of the committee has not helped. A given person will normally trust other people who they know and with whom they have built trust, and this "natural" form of trust is therefore limited to the number of people that person is comfortable with, as suggested by Dunbar's Number. There is some debate as to the exact meaning, value, and applicability of Dunbar's number, but the current 500+ membership of the C++ Standards Committee exceeds the largest estimates of Dunbar's number, in contrast to the roughly 50 committee members back in 2025.

Fortunately, there are some ways around this limit:

1. People are generally willing to trust roles, for example, most of us happily entrust our money to bank tellers and cashiers. We therefore need to take care to select trustworthy people for key

roles such as committee chairs. The increasing practice of having backup chairs and co-chairs is a most excellent step in this direction.

2. We can model and reward trustworthiness in the hope that the rest of the committee will follow. But the committee is large enough that there will be several “sneaky” people among us.
3. We can attempt to stymie people who prove to be untrustworthy, for example, by putting good rules in place. This is harder than it sounds because:
    a. One person’s untrustworthy act is another person’s valiant bending of the rules (but just a little bit) in order to make the right thing happen.
    b. Effective leadership (including committee chairs) surprisingly often requires socialization of what would otherwise be untrustworthy acts. The canonical example is a leader motivating people by promising a better future that might or might not arrive. These situations require some flexibility in rules.
    c. We can count on our smart, motivated, and creative fellow C++ committee members to game whatever rules we put in place, and the more flexible the rules, the easier the gaming.

But what is life without a challenge?

A WG can spend meetings on a proposal, carefully crafting a compromise, balancing concerns, just to have the next WG spend several meetings reviewing the proposal, demanding changes, and rephrasing the wording with patchy understanding of the design rationale as it evolved over time. In particular

- large feature developments should be discussed in joint EWG+CWG, LEWG+LWG, and/or EWG+LEWG sessions as early as the overall design and the understanding of the underlying implementation alternatives allows.

Finally, after years of process, someone then stands up in full committee and raises issues that have been discussed for years stating “lack of comfort” with the proposal, suggesting alternative approaches, and demanding more time to consider or reject. At this point, everybody unhappy with compromises made along the way chirps in with counter-points made over the years and the proposal is either withdrawn or defeated by a 20% minority, many of whom did not take part in previous discussions. We think that

- “lack of comfort” is not sufficient to block a proposal

Even when (as is common) there are minor remaining issues, typically, there are many months between a proposal being approved (by any part of the committee) and the final standard during which non-fatal flaws can be fixed. We think that a proposal that has passed a committee (say EWG or LEWG) should be accepted without undue delays by further groups. Obviously, new inputs should be considered, but in the absence of new information, “we considered and discussed that” by a WG chair should be conclusive:

- People who were not sufficiently motivated to take part in the discussion should feel obliged to at least stay neutral. Similarly, if your position has been discussed and voted down in WG, then you should at least vote neutral in plenary, to at least abide by the group’s position.

Unfortunately, this principle is easily gamed by people who break it to block proposals they don’t like. This has been observed in the committee and has led to resentment and erosion of trust.

At a first approximation,

- No proposal is ever perfect.
- Different people have very different ideas of what degree of perfection is needed for a proposal to be part of the standard (or part of a TS).

“The last bug” is a common programmer’s joke and our language specification reads like a program in a poorly defined language (English) without advanced control structures, without abstraction mechanisms, and without a compiler. Consequently, perfection will always elude us and we should take that into account when we decide what is “good enough” for a standard (or a whitepaper).

We have seen proposals move forward with issues lists to be considered before a final vote or even after. We strongly encourage that approach to make progress

- If a proposal is fundamentally sound, it should be moved forward, even into the WP text
- Refinement of text can (and should) happen later
- There are problems that are unlikely to be found until after the wording for a proposal is integrated into the WP text (as in software development: “integrate early to allow testing”)
- Addition of desired but incompletely-developed features can wait until later, even to a later standard
- As the shipping date for a standard approaches, the efforts to “polish” the text and drain the issue lists should be redoubled at the expense of effort on new proposals (as is currently done)

Through delays, a good proposal can get dated, failing to benefit from years of improvements and progress in the community. Note the file system TS and networking TS each has a decade of use behind them. Similarly, **std::variant**, **std::optional**, and **std::any** have a long history as independent proposals. That wouldn’t be too bad if the reason was that significant improvements were added during the process. However, the current process tends to reduce proposals to their most conservative cores. Thus, some “novel” C++ features feel dated by the time they are accepted.

- Aim for prompt delivery followed by incremental improvements

Note that this approach can only succeed if the end-goals are reasonably clear.

- Articulate the end goals for a proposal

When triaging features for consideration, we propose that the WG chairs to put a higher priority on features that conform to these goals, but also keep a watch against features that

- Turn C++ into a radically different language
- Turn parts of C++ into a much significantly different language by providing a segregated sub-language
- Have C++ compete with every other language by adding as many as possible of their features
- Incrementally modify C++ to support a whole new “paradigm” without articulating the end goal

- 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.)

The committee members and proposal authors are a group of volunteers who may have different aims, different views on industrial practice, different views of necessary skills, or different directives from the organization they represent. As we propose features, we urge the group to maintain cordial discourse, and aim for what is best for C++ as a language and not merely what is best for your company's current direction. This sometimes means willing collaboration with people with different aims and priorities. For the most part, we have been successful. More specifically, don't oppose a proposal just because:

- it is seen as competing with your favorite proposal for time/resources
- it is not relevant to your current job
- you have not personally reviewed it
- it is not perfect (according to your principles)
- it is not coming from your friends
- it is coming from someone you have been at odds with on different subject

Remember, we are writing a standard for millions of programmers to rely on for decades, a bit of humility is in order [Stroustrup,2019b].

## 5.2 Proposal processing

WG21 does not lack for proposals. The interest is high and the number of proposals of each mailing has grown to more than 100. That's unmanageable for an individual with a day job. To stay focused on the priorities discussed in this document, we recommend that chairs focus at least 60% of their time on advancing these priorities. How that is done is entirely the chair's prerogative. For example:

- Raise the barrier for repeated presentation of and voting on failed proposals, for example, require a written description of what changed to address previous failures.
- Require a clearly specified written rationale for each proposal.
- Require a short written tutorial for each proposal.

For any non-trivial proposal, it is not obvious how to use the new facility well in combination with other features. It is easy to say "teaching this is easy" but such statements should be backed-up with examples and experience reports (or be clearly marked as conjecture). In particular, for whom is learning and using a new feature easy?

- Never assume that the use of a proposed feature is easy and obvious to everybody.
- Never assume that the need for a proposed feature is obvious to everybody.

The volume of proposals is such that people are not able to track all upcoming votes. This has led to "no" votes in WGs and plenary.

With the advent of generative AI, the volume of submissions may increase further. For explicit DG guidance on the ethical use of AI in drafting proposals, copyright concerns, and the prohibition of unverified AI-generated normative wording, see **P4023R0: Strategic Direction for AI in C++**.

We ask the committee to consider process improvements that improve this communication and notification, but also ask that members actively review and discuss with others their proposal progress, so that the burden is fairly spread. At one time, Alistair Meredith did a great job of summarizing the flow and progress of each paper prior to C++11. Today this is unmanageable for one person. We have noted other forms of process improvement that has helped improved communication, and understanding the issues as we review papers. These include use of githubs, and summary notes recorded for each proposal, to save readers the bother of going through each meeting minutes to trace the history of a proposal and find the related papers. We do not necessarily enforce any particular methods, but feel some method should be considered to allow other groups to track progress:

- WGs and SGs should maintain an up-to-date brief list of the status of proposals being processed.
- WGs and SGs should - as far as possibly - announce what proposals will be up for discussion

Such lists should be easily accessible. It is not good enough to have information aimed at keeping people who are not regular members of a WG or SG posted on a hard-to-find mailing list requiring separate signup.

Remember the need for trust (§6.1):

- "I didn't have time to read the document" is not sufficient reason to oppose (provided the paper was submitted on time)
- "I don't understand" is not by itself sufficient reason to oppose (give reasons and examples why your lack of understanding is not just your own problem)

### 5.2.1 Guidance on Converging on unified proposals

For detailed operational guidelines on resolving competing designs, avoiding single-entity dialects, and the matchmaking role of WG/SG Chairs, please refer to the standalone DG directive: **P4024R0: Guidance on Building Consensus and Converging Proposals**.

In our experience, nearly all major proposals and a few medium proposals since C++11 have triggered alternative proposals. These lead to divergent consensus. In some cases, an alternative proposal has been brought forward at the last moment, only because it sometimes takes that long for people to understand the original proposal, to have a partial implementation of the original, see the other possibilities, or, very occasionally, it just takes that long to take notice. Sometimes, these alternative proposals have come with riders for scope creep.

We have seen this time and time again, e.g., with Concepts, Executors, and Contracts. Undoubtedly, we will see it again with future proposals. This is a good sign because it is an indication that the community cares deeply enough to want the best, and to have it represent the broadest community possible. In the following guidance, we are specifically referring to design differences, rather than implementation differences, though we accept some amount of implementation consideration needs to be factored in.

The problem of ratification of the first iteration usually comes as to when is the scope sufficient for a first design to be added to the Standard when there are divergent design directions:

- Which of several relatively attractive design proposals to choose and why?
- Which extra feature to add and why?
- What features of the status quo should be removed and why?
- Which feature should be modified and why?

Admittedly convergence is hard, especially in an emotionally charged environment, with ratification deadlines bearing down, where everyone now has a huge stake. We can only ask that stakeholders try hard to adhere to the following as we have observed it to work in the past because stakeholders have come with a willingness for honesty, compromise, wish for progress, a unified goal and a long-term realistic future perception, lacking ultimatums. Though it may be obvious, it is still worth stating:

- Aim for minimal intersection or union set that at least satisfy as many major stakeholders as possible. But do not aim for all possible stakeholders as that is untenable.
- Once you have competing proposals, work hard to get a minimal starting set, instead of building everything into the first iteration. This is surprisingly hard and should not be underestimated.
- Resist the urge to add a favourite rider feature in the initial iteration, especially when the door seems partly open to inject a significant change. This demands a great deal of discipline.
- Authors of competing proposals should get together to write a joint paper (or papers) comparing the proposals with key usage examples and key implementation issues.
- Always think in the long term, that this initial entry into the Standard is just there to gain initial experience with a minimal set, but know that if successful there is room for addition. Probably the best example of that is constexpr.

Examples of successful iteration improvements that we urge you to use as a template in seeing a long-term view of major proposals and how we build on previous work follows. In all cases, we have found a collaborative and common vision helps to soothe individual egos.

- Templates (simple non-member templates -> member templates -> additional template kinds -> concepts)
- Lambdas (non-polymorphic -> polymorphic -> constant-evaluated ->? recursive),
- Constexpr (single return statement -> general statements -> dynamic allocation & more ->? reflection).
- In many ways this also applies to library components (e.g., concurrency facilities).

Note that some differences are irreconcilable. In such cases, the committee has to make choices. We are not asking people to create bloated messes just to plaster over fundamental differences. C++ cannot be all things to all people.

### 5.2.2 ABI Stability

We believe that ABI breakage, both in the language and the library, should be considered on the merits of each individual proposal which necessitates such a break. We do not believe the labeling of any specific C++ standard as “your chance to break ABI” to be healthy for the future of C++. Such a label would encourage ABI breaks which may not bring a correspondingly high benefits to offset the cost of said ABI break. [P1654](P1654) contains some additional questions, observations and summaries of past ABI breaks (see, in particular, “Section 3: Past ABI breaks”).

The ABI Review Group (ARG) has been set up to help analyse the potential impact of ABI breaking proposals. Given the impact ABI changes can have on various implementations, such a board is in the best position to evaluate the potential consequences of such changes and the effort required to mitigate them or transition across those changes. The board makes recommendations, and as a group it also prevents any one single implementation from over-weighing their influence. As earlier versions of this paper suggested the board therefore includes at most two representatives for any product/company (one for the core language and/or one for the standard library); that does not exclude input from other people, of course: representatives are encouraged to consult with the experts they know.

### 5.2.3 Safety and Security

Safety and security remain the paramount priorities for the evolution of C++. To address this within our established committee structure— **SG23 (Safety and Security)**—the Direction Group has officially endorsed the 'Profiles' framework as the path forward. For the DG's explicit call to action—detailing the required initial profiles (Initialization, Ranges, Resources) and our request for industry funding and implementation—please refer to the standalone directive: **P3970r0:Profiles and Safety: a call to action**."

### 5.2.4 C/C++ Liaison

Similarly, we were instrumental in motivating a C/C++ Liaison SG (SG22 on the WG21 side) to improve on C and C++ cross compatibility and actively review existing and future proposals with such aim. This group has a presence in both WG21 and WG14 to avoid repeating some infelicitous divergences of the past (e.g., the incompatibility between C and C++ “inline function” mechanisms).

## 5.3 The Role of Whitepapers

Technical Specifications (TSs) were popular, but due to ISO editorial requirements, have since been replaced by ISO whitepapers. In theory, they provide an intermediate stage where major new features can be discussed, specified, tested under fewer constraints than for the standard itself, and later be moved into the standard with changes based on the experiences gained. The move from a whitepaper to an IS is supposedly simplified by the significant work to complete the whitepaper and the experience gained.

In practice, this seems to work reasonably well for libraries, though we see examples where facilities are stripped from a proposal as being too advanced, rather than pushed forward for experimentation. For language features, the experience is less positive. It seems likely that the barrier to entry into a whitepaper will not be significantly lower than for an IS, and the effort devoted to complete a whitepaper detracts from experimentation, freezing the language feature in time, years before acceptance. After the whitepaper, most of the design and specification issues are then revisited a second time. Thus, a whitepaper becomes a method for delaying a proposal. Also, a whitepaper doesn’t seem to be sufficient to encourage multiple implementations. This implies that large parts of the C++ community don't get to use the facility, large organizations using multiple compilers can’t experiment with the whitepaper at scale, and tool builders hold back waiting for a “proper standard” supported by all major implementers. Consequently, detractors can dismiss the experience reports and clamor for novel alternatives.

We recommend

- Use whitepapers for library components.
- Don’t use whitepapers for a language feature unless the feature is a mostly self-contained unit.
- Never use whitepapers simply to delay; it doesn’t simplify later decision making.
- When proposing a whitepapers, specify the “aim”: what the whitepapers is supposed to learn or achieve.
- List "exit criteria" (whitepapers to IS or whatever target) to allow people to determine whether the work is complete and whether it succeeded.
- Consider other vehicles such as SG (Study Groups), IS, and not just whitepapers
- Consider some or all the following incomplete list of frequently asked questions in your deliberations and whitepaper proposal and record their answers along with the aim and exit criteria:
    - Is there an implementation?
    - Is it a Library or Language proposal, or does it involve both aspects?
    - Is the proposal a foundational proposal, meaning many other C++ aspects/proposal depend on it, and/or it depends on many other C++ aspects/proposals?
    - Is it independent of aspects of the language?
    - Are there competing design proposals?
    - Is the proposal complicated or large that you fear there will be error in design decision?
    - Is it a research idea?
    - Is there substantial invention?
    - Can it be staged?
    - Have you refined your proposal within an open-source project such as GCC, clang, or any of a number of libraries?
    - Is there a subpart that deserves to be in IS?
    - Is the wording complicated or unconventional?
    - Will the proposal benefit from early integration (can be applied to a WP)?
    - Will you get feedback/testing only after whitepaper publication or IS publication?

- Is there a motivation to slow down a proposal?
- What would it take to turn the whitepaper into an IS?
- Are you juggling a large number of related or dependent proposals (other proposals that depend on this proposal)?
- Are you aiming for user feedback?
- Are you aiming for implementation feedback?
- Is there a scheduling concern to make C++xx for it or its dependents?

We have found some or all these questions are always asked anyway at each SG deliberation, so proposers should consider them early on. Expect the DG to offer non-binding advisory (whitepaper or IS or other target) in some cases and we hope you weigh our opinion as part of your decision process.

If a proposal isn't ready for the standard let it be improved or rejected in the appropriate WG or SG (Study Group). Don't add a formal whitepaper process.

## 5.4 "Details" vs. Design

The committee spends most of its time on details that the average programmer will never notice. A lot of that is necessary. However, it seems that most members spend most of their time on such details during the design phase; that's wasteful and weakens the language as experienced by users. Instead, early on in the process, try to spend the majority of time on design issues, such as:

- What problems is this feature meant to solve? (D&E recommends not to accept a feature that doesn't solve two apparently unrelated problems)
- What alternative solutions are possible?
- What related features should be considered simultaneously with this one?
- How well does this feature fit with the general style of the language and standard library?
- Is this feature aimed only for a few language or library experts or will it be used in application code?

An individual proposal is essentially never useful in isolation; it will work in combination with other features:

- Consider every feature in the context of features with which it will be used
- Don't approve a set of related features one by one, evaluate them together

In particular, cluster related proposals (language and library) together.

- Be fair when comparing alternative proposals

In particular, try hard to give discussions of advantages and disadvantages equal weight for each alternative. Presenting only advantages for a “favored proposal” and only “disadvantages” for an unfavored alternative is not acceptable.

During the design phase, write and maintain a tutorial for the feature. It will help others understand what the feature is supposed to achieve and help keep features aimed at “ordinary programmers” from drifting into expert-only territory. A good tutorial usually goes from the simple towards the more advanced and from the concrete to the more abstract; the tutorial in K&R is an excellent example.

The text of the standard should be precise and comprehensive. It is not supposed to be a tutorial, but the ideal is that after some “acclimatization” an experienced programmer should be able to interpret the text with some confidence. This is not currently the case. When crafting WP text

- Consider readability by people not members of the WG crafting the text
- Non-normative notes can be useful
- Short examples can be very useful
- Try not to change the meaning of terms common in the C++ community
- Where possible, use terms common in the C++ community

Remember that compiler writers and standard-library implementers are not the only target audience for the standard (or a TS).

## 5.5 Rationale, Design, and Wording

We have a standing document on how to submit a proposal:

https://isocpp.org/std/submit-a-proposal

which in turn links to a paper from 2012:

http://open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3370.html

These two papers form a great starting point on what a good initial proposal should look like. However, we have observed that the rationale and design parts are often left out of later documents containing wording and often not maintained to reflect decisions during the time taken to process the proposal. This time is typically multiple years and covers multiple meetings in multiple WGs. This can be a significant barrier to understanding and a source of confusion. For example “Is the wording in Pxxxx supposed to reflect the design in Pyyyy?” How would I know? It can be significant work finding out, often involving searching through the wikis for multiple meetings. Non-members of the committee cannot do that, so they really can’t understand what is being proposed.

It has become popular to start papers with a change record. This is welcome and useful, but not sufficient because applying multiple deltas - and often deltas to deltas - to a document requires significant effort and can be a source of misunderstanding.

We therefore suggest that the rationale and design discussions from early papers are carried through and maintained in later wording documents. This makes documents larger, but people can skip what they don’t feel the need to read. Having all in a single document also provides redundancy that can help spot mistakes. It should also help decrease the confusion that often happens when a proposal comes up for final vote so that people who have not been following the proposal closely have to catch up.

We also recommend seeking out competing ideas and implementations.  It is far better to account for these up front than to be blindsided by late objections by people who might not have been aware of your proposal.

## 5.6 Balance of concerns and engineering approach tradeoffs

Successful language design - like all successful engineering - requires fundamental good ideas and balancing constraints. Optimizing for one thing - in one direction - may succeed for one application area for one moment of time, but eventually the result dies for lack of adaptability. In our case, focusing exclusively on performance deprives us of new entrants into the community and the new ideas they bring. It also leaves us wide open to becoming blindsided by changes in hardware or optimization techniques. Super-optimized code tends to run poorly on next year's architecture and only run well on the kind of tasks for which it is optimized. This is particularly true if language facilities are modified to support specific optimizations. Furthermore, optimizations are often devoid of fundamental good ideas; they focus on tuning some status quo.

On the other hand, optimizing exclusively for ease of use can lead to very slow code. That must be avoided. There are a variety of techniques for dealing with the elegance vs. performance dilemma. One is the onion principle: there is an elegant, reasonably performant, and usually type safe  “layer” that most people can use most of the time. When more performance or more special cases need to be handled, we peel off a layer and gain more control, more performance for specific cases, and more opportunities for errors. If that’s not enough, we peel off yet another layer. This can go on until you directly  manipulate specific hardware features. The reason for referring to that as the onion principle is that “each time you peel off a layer you cry more.” Another technique for gaining both performance and simplicity is to replace standard-library components with compatible or close-to-compatible implementations tuned for a specific set of uses.

To serve its community well and thrive, C++ must address the real problems of a huge community, rather than obsess over extreme demands of a relatively small group of experts. Performance is important, but it cannot be the only concern of the community: simplicity, safety, toolability, ease of teaching, maintainability, composability of software from different sources, compilation speed, and stability are other essential concerns. There are many ways of improving performance, including cleaner code, coding guidelines, optimization hints, specialized implementations of standard components, specialized libraries, optimizers tuned for specific hardware, and specialized ABI for specific application areas.

C++ has survived for 40 years by carefully balancing concerns, learning from experience, and avoiding chasing fashions. It started up with two aims:

- utilize hardware well
- offer effective abstraction mechanisms

When given a choice between adding language features to address a problem directly and improving the abstraction mechanism, the bias was in favor of the latter. This served us well as hardware and programming techniques evolved. Furthermore, there is a constant tension between

- evolve to better address novel challenges
- stability/compatibility

When given that choice, the choice almost always was to do both. That leads to a large, more unwieldy language, but it doesn't cut users off by the knees, forcing them to choose between adopting new languages (always tempting) and avoiding costly reworking of their code.

Every language has made different engineering tradeoffs, but few have succeeded in C++'s core domains. Those language design trade-offs are engineering choices in portability vs performance vs productivity. We do not focus on any one of these to the exclusion of all others.

The nirvana of programming, one that offers world-class performance, unmatched productivity, full generality, complete safety, and ease of writing, simply does not exist. Until such a nirvana appears, it will be necessary to make engineering tradeoffs among performance, productivity, and portability.

There are a number of items that may be involved with any proposal that interacts with existing practice, such as an extension provided by one or more vendors. Firstly, consider how well the item fits into the C++ programming model. Adding inconsistent behavior would make it hard to use such a feature easily and safely. Secondly, does the proposal completely match the existing practice or are there differences? Where a similar feature is available from multiple vendors this can be particularly troubling. If there are differences, it may be better to deliberately choose a different name or syntax to avoid the risk of causing confusion (this was done, for example, with "unordered_set" because of the potential confusion with pre-existing "hash_set" extensions.)

## 5.7 C++ future online and F2F

The COVID-19 pandemic lasted for years and possibly pandemics will become more frequent, so we have adapted to a hybrid development model of being online and/or F2F.

There are some cautionary observations:

- Each week there are many WG21-created (WG and SG) meetings. Few if any can attend them all and many can't consistently attend all meetings of groups they traditionally attend in person.
- The on-line meetings have changing sets of attendees, often with proponents of new features turning up in large numbers. Often traditional “stalwarts” of a group are missing.
- For some, the steady stream of “mailings” and meetings are hard to keep up with as they disperse effort among other commitments. It can be very hard to disengage from “day job” tasks.
- The WG21 mailing lists alone produce in total between 1,000 and 3,000 messages a month, which is a large volume of traffic to keep on top of.
- The week-long face-to-face meetings allowed people to concentrate on WG21 progress — sometimes across WGs and SGs. Such concentration can be hard to achieve for many short specialized meetings. There is no time for “the cache to be loaded.”

Should a future event threaten a similar level of disruption, we offer the following lessons learned: it is hard to maintain a direction when we cannot meet F2F. Many short meetings favor progress on smaller, easier to isolate, features over major features with broad impact on the language and/or standard library.

We should be careful when we change our processes. People can't keep up when the content, logistics and process are changing very fast. We should also be very careful not to critically rely on processes and tools that some people are not familiar with and possibly cannot utilize from their work (or home) locations. There is a bewildering variety of on-line collaboration tools, video-conferencing systems, IMs, calendar systems, etc. Many would prefer to work on C++ than learning and keeping up-to-date with a shifting set of collaboration facilities and conventions.

# 6 The C++ Programmers' Bill of Rights

This note was posted to reflectors, presented at the June 2017 Toronto meeting and discussed. We propose that it be formally adopted by WG21:

## 6.1 "The C++ Programmers' Bill of Rights."

We, the ISO C++ Standards Committee, promise to deliver the following to the best of our ability, assuming user code adheres to the current standard:

1. ***Compile-time stability***: Every significant change in behavior in a new version of the standard is detectable by a compiler for the previous version.
2. ***Link-time stability***: ABI breakage is avoided except in rare cases, which will be well documented and supported by a written rationale.
3. ***Compiler performance stability***: Changes will not imply significant added compile-time costs for existing code.
4. ***Run-time Performance stability***: Changes will not imply significant added run-time costs to existing code.
5. ***Progress***: Every revision of the standard will offer improved support for some significant programming activity or community.
6. ***Simplicity***: Every revision of the standard will offer simplification for some significant programming activity.
7. ***Timeliness***: Every revision of the standard will be shipped on time according to a published schedule.

Note "to the best of our abilities". These are ideals or guiding principles, rather than executable statements. For example, if a function is added to a header file, the compilation of code that includes that header will slow down imperceptibly. That's acceptable. Adding enormous amounts of code to a header so that compilation slows noticeably would be another matter.

Note that the "improved support" in "improved support for some significant programming activity or community" can be just about anything, such as a language feature, a standard-library, a hook for tool support, a generalization of an existing set of features. It is not our aim to destabilize the language with a demand of constant change; rather to help the committee focus on what is significant to the community as opposed to insignificant changes and churn.

These are ideals. They are what we would like to see done. If we succeed, most users will be very happy. However, they are not a recipe we could blindly follow to deliver a new standard. As is typical for ideals, they can conflict: there are tensions among the desired goals. This is common: Ideally, we want quality, on-time delivery, and low cost of products, but we know from experience that it is very hard to get all three. We want freedom of speech and absence of verbal intimidation, but balancing those two can be very hard. It is the same for the ideals of the “The C++ Bill of Rights”; we want all, but the committee will have to make hard choices.

These are ideals. They are meant to be rather grand statements, rather than nit-picked long sentences watered down by adjectives and caveats. It will be up to the committee members to interpret the ideals in the light of real design and scheduling problems.

There are just seven ideals listed and they are quite general. We could add many more, but a laundry list of specifics would dull the appeal and be impossible to remember in the heat of discussions about direction, what can be delivered when, and specific technical concerns.

## 7 Caveat

The members of the direction group have been members of the standards committee for decades. We were (we think) chosen because we have deep and broad experience with the C++ technology, use, and community. We all have well-documented track records. In the role of members of the direction group, we try to serve the good of the C++ community as a whole. It is not possible to completely separate what we consider best for the C++ community from the specific proposals we work on, but we try not to unfairly favor our own proposals. Similarly, we try not to discourage proposals or actions just based on personal biases. Some of our recommendations may not be universally appreciated, but nothing we say is meant to insult anyone.

Some of the recommendations here could be seen as contradictory. We see such as fundamental tensions in a design, requiring tradeoffs, rather than contradictions. Design is hard, design by committees is even harder.

## 8 Acknowledgements

Many of the arguments and points of view have a long history in the committee. Thanks to all who contributed. In particular, thanks to the authors of the documents we reference here.

## 9 References

- [Bastien,2018] J.F. Bastien: Feedback on 2D Graphic. P1225R0. 2018-10-02.
- [Berne,2019] Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, Herb Sutter: Contracts — Use Cases. P1995R0. 2019-11-22.

- [CG,2015] C++ Core Guidelines.
- [Clow,2018] Marshall Clow, Beman Dawes, Gabriel Dos Reis, Stephan T. Lavavej, Billy O'Neal, Bjarne Stroustrup, and Jonathan Wakely: Standard Library Modules. P0581R1. 2018-2-6.
- [GDR,2016] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: A Contract Design . P0380R1. 2016-07-11.
- [GDR,2017] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: Support for contract based programming in C++. P0542R2. 2017-11-26.
- [Nishanov,2017] Gor Nishanov: C++ Extensions for Coroutines TS Document . N4680. 2017-07-30.
- [Hinnant,2017] Howard Hinnant: Extending <chrono> to Calendars and Time Zones. P0355R4. 2017-10-1.
- [McLaughin,2017] Michael McLaughlin, Herb Sutter, Jason Zink, Guy Davidson: A Proposal to Add 2D Graphics Rendering and Display to C++. P0267r6. 2017-07-30.
- [P1654] ABI WIP document. 2019-06-17.
- [Quora1] Can I use C++ for hard real-time embedded systems and kernel development?
- [Quora2] Does C-11 and C-14 make it easy to use -C++ in kernel development?
- [Saks,2016] Dan Saks "extern c: Talking to C Programmers about C++". CppCon 2016 keynote.
- [SD-8] SD-8: Standard Library Compatibility
- [StdLibGuidelines] Standard Library Guidelines
- [Stroustrup,1994] B. Stroustrup: The Design and Evolution of C++. Addison-Wesley. 1994.
- [Stroustrup,1993] B. Stroustrup: A History of C++: 1979-1991. Proc. ACM HOPL-2. 1993.
- [Stroustrup,2007] B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. Proc. ACM HOPL-III. 2007.
- [Stroustrup,2014] B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014. Addison-Wesley. ISBN 978-0321992789. 1st edition 2008.
- [Stroustrup,2015] B. Stroustrup: Thoughts about C++17. N4492. 2015-05-15.
- [Stroustrup,2015b] B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.
- [Stroustrup,2018] B. Stroustrup: NDC Keynote: What can C++ do for embedded systems developers?
- [Stroustrup,2018b] B. Stroustrup: Remember the Vasa! P0997R0.
- [Stroustrup,2018c] B. Stroustrup: The Evils of Paradigms. P0176R0.

- [Stroustrup,2019a] B. Stroustrup: C++ exceptions and alternatives. P1947. 2019-11-18.
- [Stroustrup,2019b] B. Stroustrup: How can you be so certain? P1962R0. 2019-11-18.
- [Stroustrup,2020] B. Stroustrup: Thriving in a crowded and changing world: C++ 2006-2020. P2184R0. 2020-06-10.
- [Stroustrup,2022] B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P????R0. 2022-10-15.
- [Stroustrup, 2026] P3970r0: Profiles and Safety: a call to action
- [Sutter,2018] WG21 Practices and Procedures. ISO/IEC JTC1/SC22/WG21/SD-4. 2018-01-17.
- [Winkel,2017] JC van Winkel, Jose Daniel Garcia, Ville Voutilainen, Roger Orr, Michael Wong, and Sylvain Bonnal: Operating principles for C++. P0559R0. 2017-01-31.
- [Voutilainen,2017] Ville Voutilainen: To boldly suggest an overall plan for C++20. P0592R0. 2017-02-05.
- [Wakely,2017] Jonathan Wakely: Extensions for Networking. N471. 2017-11-27.
- [Winters,2017] Titus Winters, Bjarne Stroustrup, Daveed Vandevoorde, Beman Dawes, Michael Wong, and Howard Hinnant: C++ Stability, Velocity, and Deployment Plans. P0684R0 2017-06-19.
- [Wong, 2026] P4024r0:Guidance on Converging on unified proposals
- [Wong, 2026] P4023r0: Strategic Direction for AI in C++: Governance, and Ecosystem