(Re)affirm design principles for future C++ evolution

Document Number: **d3466 R1**
Date: 2024-11-25
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: EWG

Contents

Abstract

This paper proposes that the Language Evolution WG adopt written principles to guide new proposals and their discussion, and proposes the principles in this paper as a starting point.

1 Motivation

C++ is a living language that continues to evolve. Especially with C++26 and compile-time programming, and new proposals for type and memory safety, we want to make sure C++ evolution remains as cohesive and consistent as possible so that (a) it's "still C++" in that it hews to C++'s core principles, and (b) it's delivering the highest quality value to make C++ code safer and simpler.

The Library Evolution WG has adopted written design principles to guide proposals and their discussion.

This paper proposes that the Language Evolution WG also adopt written principles to guide new proposals and their discussion, and proposes the principles in this paper as a starting point.

2 General principle: Design defaults, explicit exceptions

Notes:

- This standing document is a living document maintained by EWG. Please suggest improvements/expansions via EWG papers.
- In this document, "safety" unqualified means language safety (e.g., type and memory safety), which supports all of software security (ability to protect assets), software safety (e.g., life safety), and functional safety (e.g., bug-freedom).

The principles in this standing document design guidelines that we strongly attempt to follow. On a case by case basis we may choose to make an exception and override a guideline for good reasons, but if so we should (a) discuss and document the explicit design tradeoff rationale that causes us to consider making an exception, and (b) where possible provide a way for users to "open the hood and take control" and opt back to the design default (e.g., for a feature that may incur performance costs, provide a way to opt out if needed in a hot loop).

3 Reaffirm existing principles: [D&E] section 4.5

We should reaffirm [D&E] section 4.5's list of key principles, with minor updates indicated in [brackets].

3.1 "Retain link compatibility with C" [and previous C++]

This is under "Use traditional (dumb) linkers" but the main point that still applies.

100% seamless friction-free link compatibility with older C++ should be a default requirement. We can decide to take an ABI breaking change on a case by case basis (or, potentially, even wholesale) but should do that with explicit discussion and document the reasons why.

Example: We should not require wrappers/thunks/adapters to use a previous standard's standard library.

Example: We should not make a change we know requires an ABI break without explicitly approving it as an exception. For example, in C++11 we knew we made an API changes to `basic_string` that would require an ABI break on some implementations.

3.2 "No gratuitous incompatibilities with C" [or previous C++]

100% seamless friction-free source compatibility with older C++ must always be *available*.

Example: We can adopt "editions" or an alternate syntax, as long as there is a mode where existing syntax is available. For example, we added `using` aliases which subsumed all uses of `typedef`, and we could in future embrace a syntax mode where `using` was valid and `typedef` was rejected in that mode, but only if a mode that allowed all existing code including `typedef` was still available. (See also 2.5.)

Example: We should not bifurcate the standard library, such as to have two competing `vector` types or two `span` types (e.g., the existing type, and a different type for safe code) which would create difficulties composing code that uses the two types especially in function signatures.

3.3 "Leave no room for a lower-level language below C++ (except asm)"

Stroustrup said it well in [D&E]: "To remain a viable systems programming language, C++ must maintain C's ability to access hardware directly, to control data structure layout, and to have primitive operation and data types that map on to hardware in a one-to-one fashion. The alternative is to use C or assembler. The language design task is to isolate the low-level features and render them unnecessary for code that doesn't deal directly with system details. The aim is to protect programmers against accidental misuse without imposing undue burdens."

See also 2.5.

3.4 "What you don't use, you don't pay for (zero-overhead rule)"

And the second part: If you do use it, it's as efficient as if you had written by hand (zero-overhead *abstraction* rule).

3.5 "If in doubt, provide means for manual control"

Always let the programmer to say "trust me" and "open the hood" to take full control.

Note: This is NOT in tension with safety by default. The reverse is true: Always having a way to opt out to get full performance and control makes it feasible to make more things safe by default.

Example: C++26 makes reads of uninitialized variables be erroneous behavior by default (good safe default, no UB), but provides a way for the programmer to explicitly opt out by writing `[[indeterminate]]`.

4 Affirm additional principles

"Inside C++, there is a much smaller and cleaner language struggling to get out." — B. Stroustrup [D&E]

*"Say 10% of the size of C++… **Most of the simplification would come from generalization**." — B. Stroustrup* [HOPL-III]

This paper proposes we explicitly adopt the following additional principles.

4.1 Make features safe by default, with full performance and control always available via opt-out

We want C++ to still provide full performance and control, but not make memory unsafety mistakes easy to write frequently by mistake (initially prioritizing type, bounds, initialization, and lifetime safety, but then also progressing to other safety categories). As we add ways to make it easier to compile C++ in modes that default to "if it compiles it's safe/safer" than in the past (e.g., C++26 mode with no UB for uninitialized locals, or in a safety profile-enabled mode), we also provide ways for the programmer to explicitly say "trust me" and still use

the dangerous construct tactically where needed (e.g., by providing a syntax to suppress a bounds safety profile for one line of code in a hot loop, when the programmer has verified in other ways respects bounds safety).

Example (reprise): C++26 makes reads of uninitialized variables be erroneous behavior by default (good safe default, no UB), but provides a way for the programmer to explicitly opt out by writing `[[indeterminate]]`.

4.2 Prefer general features, avoid "narrow" special case features

We should not add special-case "narrow" features that only work in one part of the language but not others. As Stroustrup says in [HOPL-IV] section 6.1 regarding concepts, "I wanted … Full generality/expressiveness — I explicitly didn't want facilities that could express only what I could imagine."

Example: We should avoid pattern matching that doesn't work outside `inspect`/`match` expressions.

Note: It's fine to ship an initial version of a general feature that only supports a subset of use cases (e.g., as we did with `constexpr`), as long as there is a clear path to more generalized support in the language.

4.3 Prefer features that directly express intent: "what, not how"

The way to simplify code (and enable correctness, tooling, optimization, build throughput) is to enable the programmer to directly express their intent. This is the way to elevate coding idioms/patterns boilerplate into cleaner, faster, more reliable code. As Stroustrup says in [D&E] section 4.3, "Say what you mean," and elaborates in [HOPL-IV] section 2.1 as "Say what you mean (i.e., enable direct expression of higher-level ideas)."

Note: Every time we add such a feature, we make C++ much simpler to use.

Note: Where can we find these features? In existing code: Just look at the patterns and idioms that C++ code commonly writes by hand to express "how" to do a common thing, and design a way to express that intent directly as "what" they want to accomplish. (This is the same as the campus design strategy famously attributed to William H. Whyte: Build the buildings but not the paths, then wait to see where the paths go by where people wear the paths in the lawns, then pave those. We can do the same in evolving programming languages.)

Example: This includes our most popular past successes, for example range-`for` which is not only efficient but also bounds-safe *by construction* (e.g., it is easy to hoist bounds checks of a collection visited using range-`for` that directly expresses "visit each element in this range," whereas they are much harder to optimize for equivalent C++98-style loops). Similarly, `<=>` is the first feature added to the language that made the standard itself smaller and simpler, because it removed more pages of boilerplate in the library clauses than the text it added in the language clauses. Similarly: Lambda functions, `constexpr` functions, much more.

Example: `constexpr` functions directly express intent, and are simpler, safer, and much faster to build than equivalent template metaprograms that (ab)use the template type system as an accidental Turing-complete functional language to indirectly express computation, which it was never designed for.

4.4 Adoptability: Avoid viral annotation

Example, "viral downward": We should avoid a requirement of the form "I can't use it on this function/class without first using it on all the functions/classes it uses." That would require bottom-up adoption, and is difficult to adopt at scale in any language. For example, we should avoid requiring a `safe` or `pure` function annotation that has the semantics that a `safe` or `pure` function can only call other `safe` or `pure` functions. We already have `constexpr`; the more we add, the more combinatorial decorations people will need to write especially in lower-level libraries, and we should be careful that any new ones carry their weight.

Example, "viral upward": We should avoid a requirement of the form "I can't use it on this function/class without requiring all the functions/classes that use it to add it too." For example, Java checked exception annotations require listing the types that a function can throw (see also 3.5), which creates a usability barrier for every caller of that function to include those lists in its checked exception type list too; in practice the feature is not widely used, and programmers effectively opt out and disable it by writing `throws Exception` on callers.

4.5 Adoptability: Avoid heavy annotation

"Heavy" means something like "more than 1 annotation per 1,000 lines of code." Even when they provide significant advantages, such annotation-based systems have never been successfully adopted at scale, except sometimes intra-company when forced by top-down executive mandate (e.g., when Microsoft internally required SAL annotations be used; but external customers haven't adopted SAL at scale).

4.6 Avoid features that leak implementation details by default

This breaks encapsulation, which should never be the default or easy to do accidentally. It creates dependencies that makes code more brittle and harder to maintain.

Example: Java checked exceptions (again) expose implementation detail by listing the exception types that can be thrown, which may not be meaningful types at this level of the code (if they originate much lower).

Example: [P3294R1] source code (token) generation *by default* generates real source code, so it does not bypass some language rules (e.g., accessibility checking) which would mean generating a divergent language and not C++ code the programmer could write by hand. See also [P3437R0]. [P2996R5] has "splices" as form of code generation that does bypass access control (as pointers to members already do), and that is fine as long as it is not the default and easy to use accidentally.

4.7 Prefer `consteval` libraries when they can be of equivalent usability, expressiveness, and performance as baked-in language features

This enables faster iteration and higher quality features, because you don't have to be a compiler writer (or wait for a compiler update) to have the feature, and implementation in actual C++ code is easier to test and debug.

Example: We should not add a feature to the language if it could be `consteval` library code using compile-time functions, reflection, and/or generation, with similar usability, expressiveness, and performance.

5 References

[D&E] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[P2996R5] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, D. Vandevoorde. "Reflection for C++26" (WG21 paper, August 2024).

[P3294R1] A. Alexandrescu, B. Revzin, D. Vandevoorde. "Code injection with token sequences" (WG21 paper, July 2024).

[P3437R0] H. Sutter. "Proposed default principles: Reflect C++, generate C++" (WG21 paper, October 2024).

[HOPL-II] B. Stroustrup. “A History of C++: 1979-1991” (ACM History of Programming Languages conference II, April 1993).

[HOPL-III] B. Stroustrup. “Evolving a language in and for the real world: C++ 1991-2006” (ACM History of Programming Languages conference III, June 2007).

[HOPL-IV] B. Stroustrup: “Thriving in a crowded and changing world: C++ 2006-2020” (ACM History of Programming Languages conference IV, June 2020).