Too Reluctant to Break?

Backwards Compatibility Across Standards and the Relevance of C++

Stefan Kamphuis

Essay – Learn software design using C++ from the inventor of C++

Programme: International Semester in Software Engineering

Supervisor: Bjarne Stroustrup

Department of Computer Science, Aarhus University

Submitted: 30 September 2026

Word count: 2340

Contents

1 Introduction 1

2 Background 1

3 Modern C++ Is Opt-In 2

4 ABI Stability Limits Standard Library Performance 4

5 Counterarguments 5

6 Conclusion 6

References 8

1 Introduction

The first C++ release was more than 40 years ago, and billions of lines of C++ are in use today. Each new standard must keep that code working (Stroustrup, 2023 p. 260). Stroustrup calls this stability a major feature of the language (Stroustrup, 2023 p. 256). At the same time, some committee members argue that this commitment blocks improvements in safety and performance (Romeo, 2020; Winters, 2020a). Romeo (2020 section 3) suggests that C++ is becoming less attractive for new projects while remaining a practical choice for work that depends on existing code. He makes this claim only as one point in a list, without explaining in detail how compatibility leads to it.

This essay argues that C++'s commitment to backwards compatibility across standards makes it a weaker but not an unreasonable choice for new projects. This is because backwards compatibility is beneficial for existing code, despite having downsides for new C++ projects. This essay first defines the key terms and explains how C++ evolves. It then examines the downsides of source and binary compatibility and answers the strongest objection before concluding.

2 Background

Before turning to the main question, backwards compatibility across standards needs to be defined. C++ is defined by an ISO standard, revised every three years since C++11 (Stroustrup, 2023 p. 260). Backwards compatibility across standards refers to the principle that a valid program written for an earlier C++ standard still compiles, and retains the same meaning, under a later standard.

Relevance also needs to be defined. In this essay, relevance does not refer to popularity. It could mean two things: whether C++ remains a reasonable choice for existing C++ projects, as opposed to migrating them to another language, and whether C++ is a reasonable choice for new projects. Backwards compatibility clearly supports the first. The billions of lines already in use keep working under each new standard (Stroustrup, 2023 p. 260), so teams have little need to migrate them to another language. This essay

asks whether backwards compatibility affects C++'s position as a reasonable choice for new projects.

The C++ Standards Committee places great emphasis on preserving existing code by avoiding incompatible changes. For this reason, the committee rarely removes old features and only does so after years of warnings (Stroustrup, 2023 p. 267). A deprecated feature signals its eventual removal. Using it usually triggers compiler warnings, and any code relying on it will become invalid once the feature is fully removed. When features are deprecated, the committee aims to provide a transition path, usually a newer alternative that programmers can move to. This has been one of the "rules of thumb" for the design of C++ since the early years (Stroustrup, 2007 p. 4-1). For example, `auto_ptr` was deprecated and replaced by `unique_ptr` in C++11, and removed in C++17 (Stroustrup, 2023 p. 267). The six years between deprecation and removal gave programmers time to move to `unique_ptr`.

However, for compatibility reasons, deprecated features are often supported indefinitely rather than removed. Even removed features tend to still be supported by compiler vendors because their users want to keep using them (Stroustrup, 2023 p. 267). Because old features are so seldom removed, improvements usually arrive as additions alongside them, which is why old and new features coexist in the language (Stroustrup, 2023 p. 267).

3 Modern C++ Is Opt-In

The coexistence of old and new features, described in Section 2, means that each improvement makes C++ larger, even when its purpose is to make the language simpler. Features that are removed stay in the language for years after they become redundant. For example, the removal of `auto_ptr` in C++17 came six years after `unique_ptr` had superseded it (Stroustrup, 2023 p. 267). Most replacements never get that far. For example, `using` supersedes `typedef`, and lambda expressions supersede `std::bind`, yet the older features remain (Romeo, 2020 section 3). A new feature may simplify programs that use it consistently but widens the language everyone must learn (Romeo, 2020 section 3). Because the older features remain valid, knowing only the modern features is not enough. Anyone who reads or maintains C++ code must understand both.

Source compatibility does more than keep old features alive. It also means that C++ cannot fix a default, even one known to be unsafe. This is because a later standard must not change what an existing program means (Sutter, n.d.). Many dangerous defaults, such as implicit conversions and uninitialized variables, are inherited from C (Romeo, 2020 section 3). Changing them would invalidate the code of critical systems and prevent those systems from moving to a newer C++ standard (Romeo, 2020 section 4). All code therefore inherits these defaults, and staying safe depends on the programmer or on external tools catching their mistakes.

The people who support C++ the most also agree with this. Their answer is to change how C++ is used rather than the language itself (Stroustrup, 2025 section 6). The committee's Direction Group accepts that full type and resource safety is impossible without breaking backwards compatibility, which is why programmers must rely on external tools to catch type and resource errors (Hinnant et al., 2022 pp. 6-7). Stroustrup's proposed profiles would allow a programmer to ban uninitialized objects and some dangerous conversions (Stroustrup, 2025 section 6.4). However, like most new C++ features, profiles would be opt-in.

Careful programmers and tools can avoid these defaults, but nothing requires them to. Stroustrup (2025 section 6.4) admits that guidelines are essentially impossible to follow consistently in a large code base, which is why he argues that enforcement is essential. Yet the profiles meant to provide that enforcement are still only experimental and incomplete. Old code also often pulls new code back, because its outdated interfaces encourage older styles of use (Stroustrup, 2025 section 6). Safety therefore depends on each project's discipline and choice of tools, not on a guarantee from the language.

The safety changes adopted in C++26 that make existing code safer without modifying it define behavior for certain operations that were previously undefined (Sutter, 2025). Yet they still allow the programmer to opt out of the new behavior and keep the previous behavior instead (Language Evolution Working Group, 2024 section 4.1; Sutter, 2025). Standard library hardening likewise turns some undefined behavior in the library into errors that are checked while the program runs (Varlamov & Dionne, 2025 section 1). Such changes are possible because the standard never gave undefined behavior a meaning, so defining it breaks no promise, whereas changing defined

behavior would break existing code (Köppe, 2024). For example, some implicit conversions, such as passing a `double` where a `float` is expected, are defined for in-range values (ISO/IEC, 2024 [conv.double]). Only the overflow case, which is undefined, could be given new behavior.

This cost falls mainly on new projects. Backwards compatibility protects code that already exists, since it keeps working under each new standard. A new project has no old code to protect, yet it still inherits the old defaults and must rely on its own guidelines and tools to stay safe. This matters more now that safety affects which language a project chooses. The US National Security Agency advises organizations to consider moving from languages with little built-in memory protection, such as C and C++, to memory-safe languages where possible (National Security Agency, 2023). Source compatibility therefore weakens C++ as a choice for new projects.

4 ABI Stability Limits Standard Library Performance

Binary compatibility also protects old code at the expense of new projects, this time in performance. Many C++ programmers rely on a stable ABI when using precompiled code (Winters, 2020a pp. 1-2). ABI, short for Application Binary Interface, is a set of platform-defined rules about the layout of compiled code and how its parts interact. For example, the ABI rules decide how big a `std::unordered_map` is, where each piece of data goes and how functions are called. If two pieces of compiled code follow different rules, they disagree about how a `std::unordered_map` is arranged in memory. One side may read or write data in the wrong location. Nothing warns the programmer when this happens. The mismatch only shows up at runtime, as a crash, corrupted data, or wrong results. This is why old libraries and new code need the same ABI, and why changing it means recompiling everything (Winters, 2020b).

Recompiling is not always possible because the source may be unavailable. This is why improved designs for `std::unordered_map` are not adopted. Every library in the ecosystem would have to be recompiled. Winters believes that an improved design could increase performance by 200–300% in most situations. He estimates that overall, ABI constraints cost C++ 5–10% in performance. While this is not critical for most programs, it still hurts a language that people choose for its speed (Winters, 2020a).

Breaking the ABI would be even more expensive. According to Winters (2020a p. 2), implementers found the ABI breaks in C++11 expensive and painful. Even at Google, where nearly all server code is built from source, a recent ABI-breaking change cost five to ten years of engineering work. Across the whole ecosystem, Winters estimates a break would cost at least thousands of years of engineering work. In 2020, the C++ Standards Committee decided not to guarantee ABI stability forever and was willing to consider targeted breaks, but it was not ready for a broad break across the standard library (Sutter, 2020). Since a break becomes more expensive the longer it is delayed (Winters, 2020a p. 2), each delay makes the next delay more likely.

Projects that need more speed can replace parts of the standard library, but they give up the compatibility and stability it provides by turning to non-standard libraries. However, such libraries are unable to remove costs that come from the language itself, such as the overhead of passing a `unique_ptr` (Winters, 2020a p. 2).

As with source compatibility, the benefit and the cost fall on different code. ABI stability protects compiled code, which roughly half of C++ developers depend on (Winters, 2020a pp. 1-2). A new project pays the performance cost but benefits when using precompiled code. A new project that compiles all its code from source, however, pays the cost without receiving any of the benefits.

The committee's inability to reach a clear decision on the ABI has led some C++ developers to explore an alternative (Carbon Language Project, n.d.). Carbon is still an experiment. It does not prove that people are leaving C++. It only shows that some developers see the problem as serious. Winters (2020a p. 2) suggests that if the ABI is never broken, the standard library becomes what programmers use for compatibility and stability, not for performance. Binary compatibility therefore makes the language a weaker choice for new projects that prioritize performance.

5 Counterarguments

The strongest objection is that compatibility is also a promise to new projects. Stroustrup (2025 section 1) describes stability, meaning compatibility with earlier versions of C++, as "immensely important, especially for organizations that maintain

software systems for decades." A project started today can rely on the same guarantee, and it also becomes part of "an ecosystem consisting of implementations, libraries, tools, teaching, and more" (Stroustrup, 2025 section 6). He adds that tools enforcing rules such as initialization and range checking already exist and are affordable in large code bases (Stroustrup, 2025 section 6.4). For projects that are expected to run for decades, or that build on existing C++, this promise can outweigh the costs described in this essay.

However, it only pays off as the code in a project ages, while the downsides apply from the start. Source compatibility keeps unsafe defaults (Section 3), and binary compatibility prevents faster designs (Section 4). The tools Stroustrup mentions can catch the mistakes the unsafe defaults allow, but they do not remove the defaults. As with most safety measures in C++ (Romeo, 2020 section 8), these tools only check code when a project chooses to use them (Varlamov & Dionne, 2025 section 7). They also differ between implementations (Varlamov & Dionne, 2025 section 3). Even range checking cannot yet be required in the code itself (Müller, 2025). Profiles, which would offer a standard way to require such checks, were not ready for C++26 (Müller, 2025). Safety therefore remains each project's own choice, made without standard support.

The argument also has limitations. Winters's performance figures are his own estimates, based on Google's workloads and benchmarks (Winters, 2020b). Google builds nearly all its code from source (Winters, 2020a p. 2), so the ABI cost may be smaller for a typical project. This essay also includes no source that measures which languages new projects actually choose, so it shows how backwards compatibility could affect C++'s position as a reasonable choice for new projects, not that programmers are switching.

6 Conclusion

Backwards compatibility makes C++ a weaker choice for new projects, but not an unreasonable one. It benefits code that already exists but has downsides for code that has not been written yet. Source compatibility keeps old features alive and unsafe defaults in place, so a new project must rely on its own guidelines and tools to stay safe. The C++26 safety changes only improve behavior that was previously undefined. Binary compatibility keeps the standard library slower than known designs allow.

Projects can replace it with non-standard libraries, but they then give up its compatibility and stability, and the overhead that comes from the language itself remains. For projects that are expected to run for decades, or that build on existing C++, the promise of stability can outweigh these costs. For a new project written from scratch, especially one that chooses C++ for its speed, the costs weigh more heavily.

As long as the committee does not break old code to change defined behavior or the ABI, the gap between what C++ promises existing code and what it offers new code is likely to grow. Standard profiles or a planned ABI break could narrow that gap. Until then, backwards compatibility keeps C++ strongest as a language for continuing existing work, not for starting new work.

References

Carbon Language Project n.d., Difficulties improving C++, GitHub. Available: https://github.com/carbon-language/carbon-lang/blob/trunk/docs/project/difficulties_improving_cpp.md [29-09-2026].

HINNANT, H. et al. 2022, Direction for ISO C++, P2000R4. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2000r4.pdf [29-09-2026].

ISO/IEC 2024, Programming languages — C++, 7th edn, ISO/IEC 14882:2024, ISO/IEC.

KÖPPE, T. 2024, Erroneous behaviour for uninitialized reads, P2795R5. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html [28-09-2026].

Language Evolution Working Group 2024, Language Evolution (EWG) Principles, SD-10, Standard C++. Available: https://isocpp.org/std/standing-documents/sd-10-language-evolution-principles [28-09-2026].

MÜLLER, J. 2025, "Trip Report: Winter ISO C++ Meeting in Hagenberg, Austria", Developer blog, thinkcell. Available: https://www.think-cell.com/en/career/devblog/trip-report-winter-iso-cpp-meeting-in-hagenberg-austria [29-09-2026].

National Security Agency 2023, Software Memory Safety, Ver. 1.1, U/OO/219936-22, National Security Agency. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/1/CSI_SOFTWARE_MEMORY_SAFETY.PDF [29-09-2026].

ROMEO, V. 2020, Epochs: a backward-compatible language evolution mechanism, P1881R1. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1881r1.html [25-09-2026].

STROUSTRUP, B. 2007, "Evolving a language in and for the real world: C++ 1991-2006" in HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, ACM, New York, pp. 4-1.

STROUSTRUP, B. 2023, A Tour of C++, 3rd edn, Addison-Wesley, Boston.

STROUSTRUP, B. 2025, "21st Century C++", BLOG@CACM, Communications of the ACM. Available: https://cacm.acm.org/blogcacm/21st-century-c/ [28-09-2026].

SUTTER, H. n.d., Overview: What are Cpp2 and cppfront? How do I get and build cppfront?, Cpp2 and cppfront. Available: https://hsutter.github.io/cppfront/welcome/overview/ [26-09-2026].

SUTTER, H. 2020, "Trip report: Winter ISO C++ standards meeting (Prague)", Sutter's Mill. Available: https://herbsutter.com/2020/02/15/trip-report-winter-iso-c-standards-meeting-prague/ [29-09-2026].

SUTTER, H. 2025, "Crate-training Tiamat, un-calling Cthulhu: Taming the UB monsters in C++", Sutter's Mill. Available: https://herbsutter.com/2025/03/30/crate-training-tiamat-un-calling-cthulhutaming-the-ub-monsters-in-c/ [28-09-2026].

VARLAMOV, K. & DIONNE, L. 2025, Standard library hardening, P3471R4. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3471r4.html [29-09-2026].

WINTERS, T. 2020a, ABI - Now or Never, P1863R1. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1863r1.pdf [29-09-2026].

WINTERS, T. 2020b, What is ABI, and What Should WG21 Do About It?, P2028R0. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2028r0.pdf [29-09-2026].