ISO/IEC JTC1/SC22/WG21

P3866R0

Audience: LEWG

V2: An Evolution Path for the Standard Library

David Sankel [1]

[1] Adobe dsankel@adobe.com

October 6, 2025

Abstract

The C++ standard library's commitment to strong backward compatibility is a core strength, providing users with low-cost upgrades. However, this guarantee complicates library evolution. When existing types or functions require revision, compatibility constraints often result in awkward compromises, such as placing superior replacements in different namespaces or assigning them less intuitive names. This naming challenge, in turn, often discourages the introduction of otherwise beneficial enhancements. The resulting haphazard approach confuses developers about best practices and unnecessarily increases the library's perceived complexity. We propose a structured methodology for evolving the standard library that accommodates interface-breaking changes while preserving backward compatibility, thereby eliminating the need for inconsistent naming and namespacing conventions.

Historical evolution		**If V2 were adopted**	
Old	**New**	**Old**	**New**
`std::lock_guard x(m);`	`std::scoped_lock x(m);`	`std::lock_guard x(m);` `std::cpp11::lock_guard y(m);`	`std::lock_guard2 x(m);` `std::cpp17::lock_guard y(m);`

`std::sort(`
`  v.begin(),`
`  v.end());`
	`std::ranges::sort(`
	`  v.begin(),`
	`  v.end());`
		`std::sort(`
		`  v.begin(),`
		`  v.end());`
		`std::cpp17::sort(`
		`  v.begin(),`
		`  v.end());`
			`std::sort2(`
			`  v.begin(),`
			`  v.end());`
			`std::cpp20::sort(`
			`  v.begin(),`
			`  v.end());`

`std::function f =`
`  /*...*/;`
	`std::copyable_function`
	`  f = /*...*/;`
		`std::function f =`
		`  /*...*/;`
		`std::cpp11::function g =`
		`  /*...*/;`
			`std::function2 f =`
			`  /*...*/;`
			`std::cpp26::function g =`
			`  /*...*/;`

`std::thread t =`
`  /*...*/;`
	`std::jthread t =`
	`  /*...*/;`
		`std::thread t =`
		`  /*...*/;`
		`std::cpp11::thread u =`
		`  /*...*/;`
			`std::thread2 t =`
			`  /*...*/;`
			`std::cpp20::thread u =`
			`  /*...*/;`

1. Introduction

The C++ standard library's strength—its commitment to backward compatibility—is also a primary obstacle to its evolution. When components require interface-breaking changes, the committee resorts to ad-hoc solutions: placing new versions in different namespaces (`std::ranges::sort`) or giving them entirely new names (`std::copyable_function`).

The result is an incoherent library that confuses developers and harms teachability. Worse, this naming challenge actively discourages necessary improvements, as finding an acceptable name is often considered too difficult.

This paper proposes V2, a structured framework to resolve this problem.

2. V2

V2 is a proposed versioning strategy for the C++ standard library designed to manage breaking changes. This convention dictates that when a library component, such as `std::foo`, undergoes an interface-breaking revision, a new version is introduced with an incrementing numerical suffix. For instance, the first breaking change to `std::foo` would result in a new component named `std::foo2`, with a subsequent revision being named `std::foo3`.

To complement the versioned entities in the `std` namespace, we propose introducing standard-specific namespaces (e.g., `std::cpp29`, `std::cpp32`). These namespaces would contain aliases that always point to the most recent version of every component available in that C++ standard.

The recommended practice would be for developers to use these versioned namespaces by default (e.g., via `namespace cpp = std::cpp29;`). The global `std::` namespace would then be reserved for cases where direct access to an older, superseded API is explicitly required.

```
namespace cpp = std::cpp29;

void f() {
  cpp::string s; // std::cpp29::string is an alias
                 // to std::string
  cpp::unordered_map<int,int> s;
    // std::cpp29::unordered_map is an alias to
    // std::unordered_map2, a hypothetical modern
    // hash map API.
}
```

3. Associated policy

A version increment applies for changes that:

- Break source compatibility (i.e., cause compilation errors for users).
- Break ABI compatibility on major platforms.
- Introduce behavioral regressions or semantic changes that alter existing functionality.

Version increments for a component should be rare and require strong justification. Before proceeding with a breaking change, the following factors must be considered:

- **Migration Path**: Is there a feasible path for users to automatically or semi-automatically upgrade their codebase? A clear and automatable migration strategy is highly desirable.
- **Conceptual Consistency**: Does the revised component maintain its original conceptual identity? A change should not be so drastic that it becomes difficult for users to adapt their mental model of its purpose and usage.
- **Cost-Benefit Analysis**: Do the benefits of the change clearly outweigh the associated costs? This includes the engineering effort for users to upgrade, the educational burden of learning a new interface, and the added complexity of introducing another version to the standard library.

Non-breaking changes—such as bug fixes, performance enhancements, or backward-compatible feature additions—should be applied directly to the latest version of a component. The decision to backport these changes to prior versions can be made on a case-by-case basis. However, it is strongly recommended to apply such improvements to all relevant prior versions to benefit the widest possible audience.

4. Alternatives considered

4.1. Inline namespaces

Inline namespaces are a C++ feature designed to manage library evolution, allowing code compiled against an older header to link with a library that provides a newer implementation. A standard library vendor could use this mechanism, toggling the active version with a compiler flag for the C++ standard.

For example, a library could define two versions of `unordered_map`:

```
// In <unordered_map> header
namespace std {
#if __cplusplus <= 202600L // Pre-C++29
  inline
#endif
  namespace v1 {
    class unordered_map {
      /* old implementation */
    };
  }

#if __cplusplus > 202600L // C++29 and later
  inline
  namespace v2 {
    class unordered_map {
      /* new, improved implementation */
    };
  }
#endif
}
```

The primary limitation of this approach is its monolithic nature. The choice of which API version is used is a global, compile-time decision controlled by a single flag (e.g., `-std=c++29`). This forces a "flag-day" upgrade, where an entire codebase must be migrated and validated against the new versions all at once. This all-or-nothing requirement creates a significant barrier to adoption for large projects.

In contrast, the V2 framework allows for granular and incremental adoption. Developers can migrate specific components at their own pace, drastically lowering the cost and risk of modernization.

4.2. `std2`, `std3`, etc. namespaces

A 2016 proposal[1] suggested reserving namespaces like `std2`, `std3`, and so on for future, non-backward-compatible revisions of the standard library.

This approach shares a key benefit with the V2 framework: it allows for granular adoption, where developers can migrate individual components to the new API at their own pace.

However, this strategy has two significant drawbacks. First, the syntax is awkward and verbose, requiring users to manage different std-like namespaces throughout their code. Second, it fails to provide a mechanism for developers to express their intent to use the "latest and best" version of a component available in a given C++ standard.

5. Wording

If there is sufficient interest in this approach, wording can be produced for SD-9[2].

6. Conclusion

The current ad-hoc approach to evolving the C++ standard library is unsustainable. It creates a confusing and inconsistent experience for developers and even discourages necessary improvements due to the difficulty of naming them. The V2 framework presented in this paper offers a clear path forward. By combining a predictable versioning scheme (`std::foo2`) with standard-specific namespaces (`std::cpp29`), it provides a consistent and scalable solution for managing breaking changes.

Bibliography

[1] Alisdair Meredith, "Reserve a New Library Namespace Future Standardization," Feb. 2016. Accessed: Feb. 10, 2016. [Online]. Available: https://wg21.link/P0180R0

[2] Inbal Levi, Ben Craig, and Fabio Fracassi, "Library Evolution Policies," Nov. 2023. Accessed: May 13, 2025. [Online]. Available: https://wg21.link/p2267r1