

C++ Standard Library Versioning

Survey of possible ways of moving the standard library forward

Document #:	P4404R0
Date	2026-09-06
Project	ISO/IEC JTC1/SC22/WG21 — Programming Language C++
Audience	Library Evolution Working Group
Reply-to	Inbal Levi <sinbal21@gmail.com>

Table of Contents

0. Revision History.....	1
1. Introduction.....	2
2. Survey of existing proposals.....	2
3. Analysis according to user impact.....	3
4. Additional Analysis Criteria.....	5
5. Illustrated examples of the proposals.....	6
6. Real-world implementation solutions (external to ISO C++ scope).....	8
7. References.....	9

0. Revision History

1.1. R0

- Initial paper, containing four sections: essence, design principles, comparison, representative cases.

1. Introduction

The discussion on progress of the standard library has been going on for many years, with different approaches suggested and considered, each with their own tradeoffs. However, none of the approaches have had a consensus among the committee.

This paper aims to survey and classifies existing approaches (and, potentially, future approaches which may also arise), aiming to be a starting point for the discussions. The paper **does not** advocate for a specific change at this point - possible outcomes for it may be for the committee to pick a direction for evolving the standard library, but an equally possibly option is achieving a wide consensus and understanding of the limitations and why it can't be done. Either way, the aim is to put our options on the table, in a clear and concise way.

Note: ISO C++ standard specifies source-level language and library behavior, and does not specify a single C++ ABI, object format, linker, loader, package manager, or vendor servicing policy. This survey therefore separates possible standard-facing mechanisms from implementation and ecosystem techniques used as supporting evidence.

It is also worth noting there has been advocacy for tooling-focused standard document, separate from the existing C++ Programming Languages one, which may be the potential target for some of the suggestions.

Disclaimer: this is an (early) attempt to collect technics and examples and classify impact. I'm not an author of the proposals or an implementor of the tooling - mistakes are possible and likely. Fixes are welcome!

2. Survey of existing proposals

The following table aims to summarize the mechanism proposed for each paper published in recent years. If there are other proposal that should be here and are missing, please feel free to reach out, and they will be added to the next version of the paper.

NOTE: this is a very concise summary, and full read of the papers (linked in references section) is recommended.

Paper	Mechanism	Impact
P1779R3 — ABI isolation for member functions	For functions defined in a class attached to a named module, removes implicit inline status, allowing implementation hiding and explicit instantiation strategies.	No syntax change for ordinary users; module/build boundary
P1863R1 — ABI: Now or Never	Frames the choice between an ABI break and longer-term stability; lists performance and conformance costs. Does not standardize a selector.	Allowing ABI-breaking policy / rebuild boundary
P1654R1 — ABI breakage: summary	Collects past and potential ABI breaks and a spectrum from never breaking to frequent breaks; stresses ABI is implementation-controlled.	Cross-category taxonomy
P1881R1 — Epochs	Opt-in module-unit source-semantics epochs; includes possible standard-library changes and aims to avoid ABI impact for epoch rules.	Syntax/dialect selection
P3866R0 — V2	Per-component numerical suffixes plus standard-release alias namespaces; discusses inline namespace and std2 alternatives.	Source-visible per-entity versioning
P2028R0 — What is ABI...?	Defines ABI surface, catalogs blocked changes, and discusses break/stability choices and possible mangling boundary.	ABI policy and binary identity
P0180R2 — Reserve a New Library Namespace	Reserves namespace space for a future, potentially incompatible standard library; discusses inline namespaces.	New library namespace
P2465R3 — std and std.compat modules	Standardizes named standard-library modules with a compatibility-oriented split for C global names. Not ABI versioning.	Module interface selection
D3092R0 — Modules ABI Requirement	Describes module initializers, module linkage, and ABI requirements created by modules.	Modules / ABI background
N4028 — Defining a Portable C++ ABI	Proposes language and standard-library ABI facilities, including extern "abi" and std::abi concepts.	Standardized ABI boundary
SD-8 — Standard Library Compatibility	States compatibility rights reserved for future standard-library evolution.	In-place compatible evolution

Papers with additional context

1. "P2137R0 - Goals and priorities for C++": expresses one set of goals and non-goals concerning stable ABI and compatibility; it is contextual position material rather than a versioning design.
2. "P2267R1 - Library Evolution Policies": procedure for defining policies, aiming to create guidelines (and potentially unity) for different parts of the standard library.

2.1 Additional considerations to be made for each approach

- What is being versioned: a component, all of std, a source dialect, a module interface, an implementation ABI, or a package?
- What is the selection granularity: expression, component, translation unit, module unit, target, process, platform release, or ecosystem?
- Can old and new types cross function, exception, allocator, RTTI, locale, iostream, thread-local, and ownership boundaries?
- Are inline/template/constexpr definitions versioned with out-of-line runtime symbols?

- How are accidental mixtures detected: distinct names, ABI tags, linker metadata, mismatch records, package IDs, runtime checks, or documentation?
- Does “latest” move on recompilation, and can users pin a specific semantic or binary version?
- How are defects, security fixes, and conformance changes backported across retained versions?
- How do modules, header inclusion, macros, and feature-test mechanisms expose the same or different declaration sets?
- Which part requires ISO wording, and which part remains vendor/platform/package policy?

3. Analysis according to user impact

Suggested here is a criterion to classify the proposed approaches. Other criteria can also be considered, and suggestions for such are described in section “4. Additional Criteria”.

The impact on users (work being done to get the change) can be defined in terms of :

1. Changes That Do Not Affect Syntax or ABI
2. Changes That Do Not Affect Syntax but Impact Binary Compatibility
3. Changes That Include Syntax Modifications

Note: this does not include compiler flags and versions updates (which will still require work from users but will be done on a higher level).

3.1. Changes that do not affect syntax or ABI

This category retains user-visible name, APIs and ABIs. It does not exclude user-visible performance or optimization changes, but assumption is that, generally speaking, these are considered allowed withing the contract between users and the standard.

3.1.1 Behavioral, conformance, security, and performance updates within the existing contract

Examples include algorithmic improvements which are not exposed to the user, fixes to implementation defects, and added symbols while preserving existing symbols and observable guarantees. This is only practical in non-exposed logic and states, and does not support exposed layouts, templates, and inline functions.

User view

```
// User code is unchanged.
#include <algorithm>
std::ranges::sort(values);

// A vendor may replace the internal algorithm while retaining the interface
// and its required behavior.
```

3.1.2 Additions in the same namespace or class

New overloads, functions, types, customization points, or feature-test macros can be added without changing existing spellings. Additions can nevertheless affect overload resolution, name lookup, address-taking, or user code that assumes a closed set of names. SD-8 records compatibility rights reserved by the standard library and is relevant background for assessing such changes.

Example of user code

```
#include <version>
#ifdef __cpp_lib_example_feature
    auto r = std::new_facility(x);
#else
    auto r = fallback(x);
#endif
```

3.1.3 ABI-preserving evolution techniques

A library can reserve storage, use indirection/PIMPL, keep stable out-of-line entry points, or add dispatch behind an existing function. These are general library-engineering techniques; they are not a standard-library version selection mechanism mandated by ISO C++. They may trade object size, allocation, indirection, optimization visibility, or implementation complexity for future evolution space.

Example of ABI-preserving library evolution

```
class widget {
public:
    void operation();
private:
    struct impl;
    impl* p_; // fixed public layout; implementation can evolve out of line
};
```

3.2. Changes that do not affect syntax but impact binary compatibility

The same source spelling may compile to a different binary contract. This category includes class-layout changes, altered inline/template definitions, changed mangling or ABI tags, exception hierarchy changes, calling-convention changes, and different build-mode representations.

3.2.1 Rebuild as one compatibility domain

One approach is to introduce the change under the existing source API and require all affected artifacts to be rebuilt coherently. The key property is not that recompilation repairs every issue automatically, but that no old binary contract is relied upon after the rebuild boundary.

Example build command

```
# Rebuild every ABI-connected target with one toolchain/library configuration
c++ -std=c++2c -c a.cpp b.cpp
c++ a.o b.o -lstdc++ -o app
```

This model is simplest in controlled source-build environments. It is constrained by closed-source dependencies, long-lived plugins, system libraries, independently deployed DLLs/DSOs, and static archives embedded in downstream products.

3.2.2 Periodic or declared ABI-break boundaries

P1654R1 records a spectrum discussed in WG21: never break ABI; break case-by-case; break at declared boundary releases; or permit breaks more frequently. P1863R1 and P2028R0 discuss the accumulated ecosystem cost and the performance/conformance opportunities blocked by ABI stability. These papers are background and policy analysis rather than a standardized version-selection facility.

Example deployment shape

```
# Conceptual vendor boundary
libstdc++.so.OLD # old binary contract
libstdc++.so.NEW # new binary contract

# Consumers select the corresponding toolchain/runtime as a unit.
```

3.2.3 Parallel ABI identities selected without source renaming

An implementation can keep source spelling stable while placing definitions in distinct inline namespaces, attaching ABI tags, or selecting declarations with configuration macros. The changed linkage identities can turn dangerous accidental interoperation into link failures and may permit old and new definitions to reside in one runtime. Coexistence remains facility-specific: exceptions, locale facets, allocators, global state, RTTI, and APIs that exchange versioned objects require explicit treatment.

Example implementation technique

```
// Simplified implementation pattern - not user-authored ISO std declarations
namespace vendor_std {
    #if USE_ABI_V2
        inline namespace abi_v2 { template<class C> class basic_string; }
    #else
        inline namespace abi_v1 { template<class C> class basic_string; }
    #endif
}
```

3.2.4 Linker/loader symbol versioning and library identity

Some object formats and loaders can associate versions with exported symbols; shared libraries can also use different SONAMEs or DLL identities. These mechanisms can preserve old entry points or route references to compatible implementations. They are platform-specific and do not solve object-layout mismatch when C++ objects cross the boundary. C++ inline and template code may also bypass the shared-library entry point.

Example platform linker configuration

```
// Conceptual export map (not C++ standard syntax)
LIB_1.0 { global: legacy_symbol; };
LIB_2.0 { global: revised_symbol; } LIB_1.0;
```

3.3. Changes that include syntax modifications

Source-visible versioning creates a distinct API spelling. It can make migration and coexistence explicit, but it necessarily changes some source, aliases, imports, or build declarations.

3.3.1 New component name

The new facility receives a distinct semantic name or a version suffix while the old entity remains available. Current practice includes conceptually improved replacements with new names; P3866R0 proposes a systematic numerical convention such as foo2, plus standard-release namespaces that alias the version current for that release.

Source-visible adoption

```
std::function<int(int)> old_form = f;
std::copyable_function<int(int)> newer_form = f; // existing distinct-name pattern

std::widget2 w; // P3866-style numerical version
```

3.3.2 Standard-release alias namespaces

Under P3866R0, namespaces such as `std::cpp29` would provide aliases to the component versions associated with a standard release. Direct numerical names permit per-component pinning, while the release namespace expresses a coherent release view. Alias semantics mean the alias is not a third independent type.

P3866R0-style illustration

```
namespace cpp = std::cpp29; // P3866R0-style proposed
cpp::unordered_map<int, int> m; // release-selected alias
std::unordered_map2<int, int> pinned; // explicit component version
```

3.3.3 New top-level standard-library namespace

P0180 reserved namespaces for potential future standardization and discusses a future library that could break backward compatibility. A `std2`-style library can redesign a broad surface while leaving `std` intact. This is a separate library namespace rather than merely an implementation ABI namespace.

Example source spelling

```
#include <std2/vector> // Example
std2::vector<int> values;
```

3.3.4 Nested version namespace / inline namespace

A visible `std::v2` or hidden implementation inline namespace can distinguish versions in mangled names. If users explicitly qualify a version, adoption is source-visible and potentially fine-grained. If a header or flag changes which namespace is inline, unqualified source can remain unchanged but the binary identity changes globally for the affected declarations.

General C++ inline-namespace pattern

```
namespace library {
  namespace v1 { struct value; }
  inline namespace v2 { struct value; }
}
library::value current;
library::v1::value legacy;
```

3.3.5 Versioned headers

A separate header can expose a revised declaration set. This makes selection explicit but requires the implementation and build graph to prevent incompatible declarations from being combined in one translation unit or across an ODR-connected program.

Example versioned header

```
#include <soot2>
soot2::tree t;

// versus legacy
#include <soot>
soot::tree old_t;
```

4. Additional Analysis Criteria

4.1 Analysis according to program layer

We should also take into consideration the layer to which the update/fix/change is done

1. Source/API - existing source compiled unchanged, and old and new spellings coexist.
2. Translation - headers, modules, macros, dialect flags, and definitions seen by each translation unit
3. Object/ABI – changes in type layout, mangled names, calling conventions, inline definitions, exceptions thrown, or templates instantiations.
4. Link/load - old and new symbols/libraries, either as an addition or as breaking change
5. Package/deployment – mechanism by which compatible binary variants identified, distributed, selected, and updated.

4.2 Analysis by engineering dimension

This analysis is done by structure and identify impact on each component of the program, compilation flow, and pipeline.

1. Source compatibility - Whether existing source remains valid without edits when recompiled.
2. ABI / binary compatibility - Whether existing binaries interoperate correctly without recompilation.
3. Version coexistence - Whether old and new versions can safely exist in one program or process.

4. Rebuild requirement - Which translation units, libraries, packages, or applications must be rebuilt.
5. Source migration requirement - Which source declarations, calls, includes, imports, or qualifiers must change.
6. Compiler involvement - Whether the compiler must implement a mode, mangling rule, module rule, or diagnostic.
7. Linker / loader involvement - Whether symbol versions, library identities, mismatch records, or loader selection are required.
8. Build system / package manager involvement - Whether build metadata must select, isolate, or propagate a version or ABI variant.
9. Modules interaction - Whether the approach changes module names, compiled module artifacts, imports, or module-unit semantics.
10. Scope - Whether the mechanism needs ISO wording, implementation support, ecosystem tooling, or a combination.

The options are not mutually exclusive. A standards-level naming convention can be combined with vendor ABI namespaces, shared-library symbol versioning, module artifacts, or package-manager variant selection.

For example, we can consider different solutions in relation to the engineering dimensions above:

Solution	Source syntax	Binary effect	Coexistence	Prim responsibility
A. Source stable update	Unchanged	None intended	Single logical version	Library implementation / servicing
B. In-place API-stable ABI break	Unchanged	Layout or symbol contract changes	Usually requires rebuild; dual ABI is possible	Toolchain, library and linker
C. Additive replacement under a new name	Use new type/function name	New symbols/types	Fine-grained	WG21 library API
D. Versioned namespace or library namespace	Qualified namespace changes or alias	Names separate versions	Fine- or release-grained	Standard wording plus implementation
E. Per-entity numerical suffix + release aliases	foo2 plus std::cppNN alias	Separate entities	Fine-grained	Standard naming policy
F. Header or module-name versioning	Different include/import	Potentially separate declarations and binaries	Unit/package-grained	Standard interface + build system
G. Dialect / epoch selection	Module/TU declaration or compiler mode	Can be none or coordinated	Translation-unit/module-grained	Language mode and implementation
H. Implementation dual/multi ABI	Usually unchanged; macro/flag selects	Parallel mangled names/layouts	Implementation-dependent	Vendor headers and runtime
I. Periodic coordinated ABI break	Usually unchanged	Broad rebuild boundary	Old/new runtime may be parallel	Vendor/platform ecosystem
J. Package-manager binary variants	Unchanged application API	Different packages/IDs	Process/environment/package-grained	Conan/build/deployment tooling

5. Illustrated examples of the proposals

Disclaimer: I'm not the author of any of these, and these were created with best intent, but mistakes are possible. Fixes are welcome!

5.1 Revised unordered associative container

Three distinct user experiences

```
// Additive/new-name route
std::unordered_map<int, int> legacy;
std::unordered_map2<int, int> revised;           // require modification in source code (may be find/replace, but API changes have larger impact)

// Release alias route
std::cpp29::unordered_map<int, int> selected;    // require minor change in source code, for example in namespace alias, then rebuild

// In-place ABI-break route
std::unordered_map<int, int> same_source;        // all consumers rebuilt
```

A revised bucket representation or hash-state layout can be source-compatible while ABI-breaking. A new entity avoids silently reinterpreting old objects but introduces conversion, naming, generic-code, and teaching questions. An in-place break retains source spelling but shifts burden to rebuild and deployment coordination.

5.2 Library update delivered through modules

Module-facing source

```
import std;          // stable standard module name
// import std.vNext; // hypothetical preview/version name

auto data = std::vector<int>{1, 2, 3};
```

The build system must provide the correct compiled module interface and order imports. If the imported declarations expose a changed vector layout, every binary consumer still needs a coherent ABI. The module name can serve as a selection point, but the language feature does not make layouts compatible.

5.3 Mixed legacy binary through an adapter

Stable adapter example

```
// Adapter compiled with the legacy ABI
extern "C" size_t legacy_transform(const char* in, size_t n,
                                  char* out, size_t cap);

// Modern code passes bytes, not either side's std::string object.
```

This isolates the standard-library type boundary and can be packaged as a dedicated compatibility component. It adds copying/ownership/error-model considerations and does not allow arbitrary old/new C++ objects to interoperate.

5.4 Dialect or epoch selection

P1881R1 proposes an opt-in module-unit epoch declaration to change language meaning while retaining compatibility across separately selected source regions, and explicitly discusses standard-library changes. It states a design goal of no ABI impact for epoch rules, but library changes that alter types or exported signatures would still require their own compatibility analysis. An epoch is therefore primarily a source-semantics selector, not an automatic library ABI solution.

Example epoch selection

```
epoch 2029;          // illustrative syntax from an unadopted proposal family
export module app;
// Source in this module unit is interpreted under the selected rules.
```

5.5 Compiler standard-mode selection

Existing `-std=` / `/std:` modes select language and library availability. Implementations commonly avoid changing established standard-library ABI solely from the language dialect flag; GCC explicitly documents that its dual-ABI choice is independent of `-std`. A future approach could coordinate a dialect and library version, but that would enlarge the rebuild and mixed-mode compatibility problem.

Existing toolchain pattern

```
c++ -std=c++20 legacy.cpp -c
c++ -std=c++2c modern.cpp -c
# Linking the objects is safe only for the contracts the implementation supports.
```

5.6 Versioned named modules or partitions

A standard-library interface could expose distinct module names or partitions, such as an unversioned compatibility module and a versioned revised module. This would be source-visible and would require build systems to select matching BMI/CMI artifacts and libraries. P2465R3 provides existing standardized names (`std` and `std.compat`) for a different compatibility split; it does not version the library ABI.

Standardized and hypothetical spellings

```
import std;          // standardized C++23 library module
// import std.v2;     // hypothetical versioned module name
```

5.7 Package / distribution version selection

Package managers can keep binary variants separate by compiler, compiler version, architecture, options, dependency versions, runtime model, or a declared ABI option. This controls distribution and graph composition; it does not change ISO library semantics. Conan 2 computes package identities from settings/options/dependencies and permits custom compatibility policies.

Example Conan profile/option

```
[settings]
compiler=gcc
compiler.version=15
compiler.cppstd=23
arch=x86_64

[options]
stdlib_abi=v2 # project-defined option
```

5.8 Stable boundary adapters

A process can isolate evolving C++ types behind a deliberately stable C or handle-based boundary, conversion layer, RPC boundary, or plugin protocol. This can permit components using different C++ libraries or ABIs to coexist without passing ABI-sensitive objects directly. It is an application/vendor architecture technique, not a replacement for standard-library evolution inside one C++ ABI domain.

Example adapter boundary

```
extern "C" {
    typedef struct text_handle* text_handle;
    text_handle text_create(const char* bytes, size_t n);
    void text_destroy(text_handle);
}
// Each side converts between the stable handle protocol and its local std::string.
```

6. Real-world implementation solutions (external to ISO C++ scope)

Note: the following examples are implementation-specific, and external to the current “C++ programming language” scope.

6.1 GCC/libstdc++ dual ABI: std::string and std::list

GCC 5.1 introduced new libstdc++ implementations of std::string and std::list to satisfy C++11 requirements: copy-on-write strings were no longer suitable, and list needed to track its size. The change is associated with GCC 5.1 and C++11 conformance, not a C++14-mandated SSO break. The new ABI is independent of the -std dialect option.

- New definitions use an inline namespace, for example std::__cxx11::list, producing distinct linkage names.
- Old and new implementations are present in the same library without changing its SONAME.
- _GLIBCXX_USE_CXX11_ABI selects which declarations headers expose for each compiled source file.
- Not every affected facility is simply duplicated; the documentation calls out locale facets and std::ios_base::failure handling.
- The mechanism enables substantial coexistence, but APIs that exchange old/new string or list objects still require consistency or an adapter.

Actual libstdc++ selection macro

```
# New ABI (default for a normally configured modern GCC)
g++ -D_GLIBCXX_USE_CXX11_ABI=1 -c new.cpp

# Compatibility build against the old ABI
g++ -D_GLIBCXX_USE_CXX11_ABI=0 -c legacy_adapter.cpp
```

6.2 libc++ build-time ABI versions and opt-in ABI flags

libc++ documents a stable default ABI and an unstable “next” ABI selected when building libc++. ABI-changing features are gated by internal macros controlled by the library ABI version; vendors, rather than ordinary users, are expected to configure them. This demonstrates a vendor build-time channel for accumulating and testing breaking improvements separately from the stable ABI.

Documented libc++ build options

```
cmake -DLIBCXX_ABI_VERSION=1 ... # stable default
cmake -DLIBCXX_ABI_UNSTABLE=ON ... # unstable ABI build
```

6.3 Microsoft STL build-mode representations

Microsoft documents _ITERATOR_DEBUG_LEVEL values that select checked/debug iterator behavior. Some modes change container and iterator representations; the Microsoft STL hardening notes describe the modes as ABI-incompatible. Link-time mismatch detection is used in supported toolchains. This is an example of configuration creating distinct ABI domains even without changing standard source names.

Actual implementation macro; simplified command

```
cl /EHsc /D_ITERATOR_DEBUG_LEVEL=0 consumer.cpp
// All ABI-connected objects must use a compatible setting.
```

6.4 Conan binary model

Conan identifies binary packages from settings, options, and requirements. Conan 2’s default model expects close matches for settings/options and applies dependency-sensitive modes; compatibility behavior is customizable. It can therefore distribute separate stdlib ABI variants, select a compatible fallback under declared policy, or fail when no matching binary exists. The package identity is metadata and selection logic, not proof of ABI compatibility.

Example Conan recipe logic

```
def package_id(self):
    # Project-defined policy: retain or normalize ABI-relevant settings.
    # Erasing information asserts compatibility; it does not establish it.
    pass
```

7. References

WG21 papers and links

P1779R3 - ABI isolation for member functions

P1654R1 - ABI breakage - summary of initial comments

P1881R1 - Epochs: a backward-compatible language evolution mechanism

P1863R1 - ABI - Now or Never

P3866R0 - V2: An Evolution Path for the Standard Library

P2028R0 - What is ABI, and What Should WG21 Do About It?

P0180R2 - Reserve a New Library Namespace for Future Standardization

P2465R3 - Standard Library Modules std and std.compat

D3092R0 - Modules ABI Requirement.

N4028 - Defining a Portable C++ ABI

SD-8 - Standard Library Compatibility.

P2267R1 - Library Evolution Policies (Governance/procedural context)

P2137R0 - Goals and priorities for C++ (Contextual position paper)

Evolution-related implementation, tooling, and ecosystem data

GCC libstdc++ Dual ABI - GCC 5.1 dual-ABI documentation.

GCC libstdc++ ABI Policy - Library and compiler ABI policy.

libc++ ABI Versioning - ABI build versions and unstable mode.

libc++ ABI Guarantees - Vendor ABI flags and guarantees.

MSVC _ITERATOR_DEBUG_LEVEL - Iterator debug mode documentation.

Microsoft STL Hardening - Implementation-specific hardening modes.

Conan Binary Compatibility - Package-ID matching and compatibility customization.

Could C++ Developers Handle an ABI Break Today? - Luis Caro Campos (Conan) (CppCon 2025)

CMake C++ Modules - Module scanning and build graph documentation.

Microsoft Standard Library Modules - Implementation guidance for std and std.compat.