C++26 Contract Assertions, Reasserted

Document #:	P3846R1
Date:	2025-11-03
Project:	Programming Language C++
Audience:	EWG

Reply-to: Timur Doumler <papers@timur.audio>
Joshua Berne <jberne4@bloomberg.net>

Gašper Ažman <gasper.azman@gmail.com>
Peter Bindels <dascandy@gmail.com>
Peter Dimov <pdimov@gmail.com>
Louis Dionne <ldionne@apple.com>
Eric Fiselier <eric@efcs.ca>
Mungo Gill <mungo.gill@me.com>
Pablo Halpern <phalpern@halpernwightsoftware.com>
Tom Honermann <tom@honermann.net>
Corentin Jabot <corentin.jabot@gmail.com>
John Lakos <jlakos@bloomberg.net>
Nevin Liber <nliber@anl.gov>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Roger Orr <rogero@howzatt.co.uk>
Nina Dinka Ranns <dinka.ranns@gmail.com>
René Ferdinand Rivera Morell <grafikrobot@gmail.com>
Oliver Rosten <oliver.rosten@gmail.com>
Iain Sandoe <iain@sandoe.co.uk>
Hui Xie <hui.xie1990@gmail.com>

Abstract

Recent papers and NB comments raise concerns about contract assertions as specified in the C++26 working draft, some even arguing for their removal. Some of the concerns are legitimate yet unavoidable with *any* viable assertion facility. Others reflect misunderstandings of the proposal, leading to inaccurate observations.

Almost all objections are repetitions of those raised in earlier papers, addressed in subsequent responses, and extensively discussed in EWG before contract assertions were incorporated — with strong consensus — into the C++26 working draft. This paper summarises the recurring objections and provides concise responses.

Contents

Revision History

Revision 1 (Update for Kona November 2025 Meeting)

- Added significant clarifications and descriptions of successful implementation experience in Concern 2
- Added Concern 18 about standard-library hardening in response to [FR-001-014], [US 3-015], [US 61-112] [FR-010-113], and [P3878R0].
- Added discussion of [P3853R0]
- Added discussion of post-October 2025 Mailing papers: [P3889R0], [P3893R0], [P3896R0], [P3909R0], [P3910R0]
- Various minor edits and clarifications

Revision 0 (October 2025 Mailing)

- Original version of the paper

1 Introduction

WG21 has been pursuing contract assertions for more than two decades ([N1613], [N4378], [P0542R5]). After contract assertions were removed from the C++20 working draft in 2019 ([P1823R0]), WG21 formed SG21 with the mandate of designing a contract-assertion facility that meets the needs of the community and has strong consensus within WG21.

SG21 spent five years gathering use cases ([P1995R1]), exploring the design space, researching alternatives, and refining every aspect of the proposal before forwarding it to EWG, where it was scrutinised for another year. The product of this work, [P2900R14], was adopted — with strong consensus ([N5007]) — into the C++26 working draft at the February 2025 meeting in Hagenberg. A comprehensive record of this effort can be found in [P2899R1]. Complete implementations exist in publicly available forks of GCC and Clang ([P3460R0]), with upstreaming in progress.

Contract assertions as specified in P2900 are defined by three characteristics: being redundant to the correctness of the program, being independently checkable, and having their evaluation semantics determined independently from the source code ([Lippincott24], [Sutter25]). These characteristics fundamentally distinguish contract assertions from every other existing C++ feature and can make reasoning about their design initially unfamiliar, even to experts.

As P2900 progressed through SG21, EWG, CWG, and Plenary, each widened audience has renewed debate, often about the same concerns raised earlier. However, once the rationale for P2900's design decisions was heard, each group reconfirmed the design, with strong consensus to move forward. NB comment resolution has again raised known concerns.

Before discussing the recurring objections, recall the value that contract assertions bring to C++26 and the reasons extraordinary effort was invested in their standardisation.

- **Consistency across libraries.** Without a standard assertion facility other than the limited C `assert`, libraries develop bespoke assertion facilities with incompatible functionality and configuration models. Managing checking and violation reporting at the application level requires detailed knowledge of the entire software stack and does not scale. Standardising an assertion facility is the only way to mitigate this complexity.
- **Improved code analysability.** Exploiting macro-based assertions such as C `assert` and its bespoke variants for static analysis is often inapplicable at scale because of the lack of a standard syntax and the need to inspect function definitions in distant translation units. C++ contract assertions provide a standard syntax for specifying preconditions and postconditions on function *declarations*, substantially improving the ability of static analysis tools to detect bugs.
- **Expressing intent more directly.** The ability to express preconditions and postconditions on function declarations allows the developer to express the intended use of a function interface *directly in code*. This easily accessible information helps human and AI readers to reason about the program and allows deeper integration with IDEs and other tools to provide an improved developer experience and reduce the probability of introducing bugs.
- **Integration with the language.** With macro-based assertions, many common cases — such as checks before the member initialiser list of a constructor, after the destruction of members and base objects in a destructor, or after the initialisation of a function's return value —

require workarounds that subtly change the meaning of code. C++26 contract assertions allow these cases to be handled correctly and without altering semantics when checks are disabled.

- **Mixed-mode support.** Macro-based assertions inevitably run afoul of the ODR when attempting to use different configurations in different translation units. C++26 contract assertions provide configurable checks without making mixed mode IFNDR. This approach enables the use of ODR checkers and makes unsound optimisations that would otherwise break the program non-conforming.

The benefits listed above are not exhaustive; for a more detailed review, see Section 2.1 of [P2899R1] and see [P3204R0].

Recent papers ([P3829R0], [P3835R0], [P3849R0], [P3851R0]) and NB comments (for a full list, see the Appendix) raise concerns over contract assertions; some argue for their removal from the C++26 working draft. Almost all the objections in these papers and NB comments repeat those that were raised in earlier papers ([P3173R0], [P3478R0], [P3506R0], [P3573R0]), addressed in subsequent responses ([P2899R1], [P3500R1], [P3578R1], [P3591R0]), and extensively discussed in EWG.

Due process requires these concerns to be heard and considered, and they have been at each stage in the past. Some reflect misunderstandings of the proposal, leading to inaccurate observations; others are legitimate concerns that are not, however, specific to P2900 and would apply equally to *any* viable assertion facility.

This paper summarises the current concerns, provides a concise description of how they have been previously considered, analysed, and addressed, and clarifies why they fail to justify removing contract assertions, one of the cornerstone features of C++26, from the working draft.

2 Discussion

We have structured the discussion of each concern into sections.

- **Summary:** Our description of the concern, with references to the papers and NB comments where it was raised
- **Discussion Status:** Whether, when, and/or where this concern was previously considered, and whether we believe any new information has been presented that was not part of earlier consideration
- **Response:** A brief explanation of why the concern, when previously considered, did not prevent contract assertions from achieving consensus
- **Details:** A more complete analysis of the concern and our response to it, suitable for those who have not participated in the general discussion within WG21, with references to existing publications where appropriate

We encourage readers to take advantage of this structure to let them skim the concerns for those where they feel the need for more information.

Concern 1: P2900 is not 'safe' or makes C++ 'less safe'

Summary

[P3835R0], [US 25-052], [FI-071], and [RO 2-056] characterise P2900 as being 'not safe' and diminishing the overall 'safety' of C++. A central concern is that P2900 provides no method to guarantee *in code* that a particular assertion, or all assertions in a given 'component of a program', will always be checked. In addition, [RO 2-056] suggests that being able to alter the evaluation semantics undermines the reliability of the feature and its successful adoption. The suggested resolutions are to add labels ([P3400R1]) to C++26, remove the *ignore* semantic, or remove P2900 from C++26 entirely.

Discussion Status

These arguments were raised repeatedly in [P3173R0], [P3362R0], [P3506R0], and [P3573R0], responded to in [P3376R0], [P3500R1], [P3578R1], and discussed in EWG on multiple occasions — most recently in Hagenberg — without producing new technical insight. The specific suggestion of changing the meaning of `pre` and `post` to always be checked was polled in EWG in Tokyo and rejected with strong consensus. No new information has been presented since.

Response

Assertions do not make C++ 'less safe'. When checked, they can detect bugs; when ignored, they have no runtime effect while documenting intent. The ability to configure their evaluation semantics externally is a prerequisite for widespread adoption, not a defect. Assertions enable developers to incrementally improve the correctness of their code but are not intended to prove or guarantee the absence of undefined behaviour on their own. Mechanisms for non-ignorable checks exist today (e.g., the `if` statement) and are not weakened by P2900. Nevertheless, non-ignorable checks can benefit from other properties of contract assertions, and P2900 provides a clear evolutionary path to support them, proposed in [P3400R1] as a post-C++26 extension.

Details

Given the recent focus on improving safety in C++ ([Sutter24]), concerns understandably arise over whether new features advance that goal. However, this debate is hampered by the ambiguity of the word 'safety' ([P3500R1], [P3578R1], [P3376R0]) and the resulting conflation of two distinct goals:

- Improving *functional safety* — the likelihood that a program performs its intended function without causing harm
- Improving *language safety* — the absence of undefined behaviour or the ability to prove such absence

Both are important and related, but they favour different priorities and thus require different solutions. Contract assertions, as specified in P2900, are designed to *express the programmer's intent* ([Meyer97]) as widely and liberally as possible (Rule 68 of [Sutter04]) across a wide range of codebases. They primarily address functional safety, not language-level guarantees. Expecting them

to do both overlooks the complimentary nature of these goals ([P3578R1]), while expecting them to serve only the latter disregards their well-documented use cases ([P1995R1]) and benefits (Section 2.1 of [P2899R1]).

Assertions — whether contract assertions, C `assert`, or any other variant — improve a program's correctness and functional safety *incrementally*. Each added assertion increases the potential to detect another bug; when *ignored*, the correctness of the program is unaffected. Assertions cannot make a program or the language 'less safe'; an assertion that does *nothing* is no worse than having no assertions at all. The ability to ignore assertions does not hinder their adoption, but is what *enables* it, as proven by decades of successful use of C `assert` and similar macro-based assertion facilities.

By contrast, *non-ignorable checks*, as sought by [P3835R0], [US 25-052], [FI-071], and [RO 2-056], serve a fundamentally different need: providing a *guarantee* that certain conditions are always verified because the risk of continuing after a violation is unacceptable. Such guarantees can be expressed today with ordinary control flow (e.g., `if` statements). Standardising contract assertions doesn't hinder the continued use of this approach.

That said, the features added by P2900 can offer advantages even for non-ignorable checks — such as the ability to place checks on declarations and to route violations through a global, link-time replaceable handler. P2900 provides the following options to achieve this:

- Use `pre` and configure the build to *always* compile it with a checked semantic.
- Duplicate the check, using both `pre` on the declaration and `if` in the definition.
- Use a vendor extension such as the attribute implemented in Clang

  ```
  pre [[clang::contract_semantic("quick_enforce")]]
  ```

- Continue to use `if` statements until post-C++26 extensions become available.

The most direct way to achieve non-ignorable contract assertions with P2900 is to control how that contract assertion is built and to choose a checked semantic when building it. For non-inline functions, that is the choice made for the translation unit in which the function is defined. For inline functions, as in the example from [P3835R0], control requires having a consistent build configuration across all translation units that compile the function. Alternatively, in some situations, force-inlining the function or giving it internal linkage is a viable strategy to reduce the decision to a single translation unit. Where users control the build configuration of all relevant translation units, they can reliably enforce a chosen semantic.

[RO 2-056] finds the current options insufficient for its use case and proposes three remedies: adopting [P3400R1] for C++26, removing *ignore* from P2900, or removing P2900 from C++26 entirely.

[P3400R1] adds *labels*, giving users the ability to control the evaluation-semantic *in code*. This is required to make optional checks more scalable and — as a side effect — makes non-ignorable checks directly expressible. This proposal is being actively pursued and has received productive feedback, but it is not yet ready to adopt and has not yet reached consensus, in any WG21 subgroup, to move forward. Inclusion in C++26 is, therefore, not an option. We do, however, expect it to be implemented soon after C++26 contract assertions ship with Clang and GCC, potentially well before C++29 ships.

Removing *ignore* from P2900 would limit contract assertions to serving *only* the specific use case [RO 2-056] seeks to address and remove their utility for most other purposes:

- In performance-sensitive code, non-ignorable assertions will be omitted unless the developer knows beforehand that the check is cheap enough.
- Otherwise, they will be removed after initial development to claw back performance, allowing regressions to creep in.
- Libraries without the ability to ignore redundant checks risk being abandoned for more performant alternatives.

None of these outcomes support widespread adoption or effective use of P2900 for its intended purpose. Some libraries may never need to consider the performance impact, but many C++ libraries begin with a focus on performance or eventually find they must have one due to the needs of their clients. Moreover, these effects lead to the least use of contract assertions in the most widely used and successful libraries — the opposite of the relationship desired for a feature that aims to reduce bugs. All these outcomes are avoided by allowing redundant checks to remain optional — with that choice made at build time, not in source code.

Finally, removing P2900 from C++26 entirely would not benefit *any* constituency; it would only block a feature that will eventually serve all of them. To achieve the latter, the pragmatic and established approach is incremental standardisation (see Concern 14): deliver the core facility in C++26 so users can begin adopting it, and extend it in C++29 to cover additional use cases such as non-ignorable checks.

Concern 2: P2900 does not provide consistent semantics across TUs

Summary

[P3829R0], [P3835R0], [P3851R0], and [ES-048] express concern that in a program comprising multiple translation units potentially compiled with different contract-evaluation semantics (also called *mixed mode*), P2900 does not guarantee that the semantic of any given contract assertion shared across TUs will be dictated by the configuration of any particular one of those TUs. The papers characterise this as a major safety issue.

Discussion Status

This concern was previously raised in [P3573R0] and addressed in [P3591R0]; EWG discussed these papers in Hagenberg. The implementability of mixed mode and the tradeoffs involved were covered in [P3267R1] and [P3321R0], which were discussed by SG15 in St. Louis and Wrocław, respectively, with implementers from EDG, GCC, and Clang present; no one expressed strong concerns regarding P2900 mixed mode from a compiler or build system perspective. No new information has been presented since.

Response

Mixing translation units compiled with different build flags is an inevitable consequence of the C++ compilation model. Any assertion facility needs to address this reality. With C `assert`, mixed mode makes programs IFNDR, with limited mitigation options in tooling due to missing information about macro values. With P2900, not only is the behaviour limited to one of the semantics incorporated into the program, but it also allows a variety of tooling approaches with different tradeoffs. One option is to enhance symbol names with information about the semantic, enabling linkers to make a deterministic choice. Even in a naive implementation, the worst case is that an assertion will go unchecked for users that incorrectly assumed they were the only one compiling a given function, which by design cannot introduce a new bug into an existing program.

Details

Consider the example shown in [P3835R0]: a shared header `z.h` with an inline function `f` containing a contract assertion. The correct choice for the evaluation semantic of such an assertion depends on many factors. For instance, if `f` is used in a performance-critical loop, anything but *ignore* may be prohibitively expensive. In higher-level code that is less performance-sensitive and less well tested, *quick-enforce* might be more appropriate. If diagnostic messages are required, *enforce* might be the correct choice. An assertion facility must support this variability; requiring uniform semantics across an entire program would result in a feature unusable at scale.

The flexible model in P2900 allows contract-evaluation semantics to vary from one evaluation of an assertion to the next and in any way the implementation chooses. For example, enforcing preconditions but ignoring postconditions is a conforming strategy; observing every tenth evaluation of an assertion and ignoring the remaining ones is another; delaying the choice of evaluation semantic until link time, load time, or run time is a third. In practice, any such strategy will be chosen at compile time — per TU — via a build flag or combination of them. A particularly natural initial strategy — already implemented in GCC and Clang (see the table in Concern 6) — is to compile each TU with a single semantic that will be used for all contract assertions in that TU.

[P3835R0] seeks 'a mechanism to make sure that a consistent semantic is provided for a given component'. The flaw in this request is that no such notion of 'component' exists in the C++ language or ecosystem, and there no consistent definition that could be applied in a specification. Introducing such a notion would require an in-source mechanism that likewise does not exist, and furthermore should not be provided for the sole purpose of contract assertions. Once such a mechanism does exist, it will be straightforward to apply it to contract assertions via labels ([P3400R1]).

In the absence of a notion of 'component' or a labels mechanism, choices apply at TU granularity. [P3835R0]'s idea that an inline function might 'belong' to a TU is inconsistent with reality — an inline function `f` belongs equally to every TU that includes `z.h`. If multiple such TUs are compiled with different build flags (mixed mode), each TU will produce its own definition of `f` with potentially different evaluation semantics for its contract assertion. Existing linkers will then have to select one version.

With a naive implementation of P2900 that does not preserve information about the chosen contract-evaluation semantic until link time, this selection happens arbitrarily. On such a naive implementation, users who do not fully control their build environment cannot reliably predict which evaluation semantic applies to non-inlined calls to `f`. This lack of control is no worse than the status quo with macro-based solutions. However, unlike macro-based solutions, P2900 opens the door to implementations improving the situation by preserving additional information until link time. Note that having a language feature *require* such QoI improvements would greatly hinder adoption, because many environments where the use of new Standards and their accompanying features is desired also lack control over the linkers they must use.

Implementation strategies for mitigating the issue highlighted by [P3835R0] include the following (none of which require any change to the specification in P2900):

- The naive implementation (implemented in GCC and Clang): compile each function with the contract-evaluation semantic specified for that TU; if multiple definitions exist, the compiler will choose one arbitrarily. The worst case (barring compiler bugs such as those described in Concern 4) is that a contract assertion intended to be checked is instead ignored, which is no worse than if contract assertions did not exist. This case will occur only when the assertion is compiled in a TU where ignore has been chosen at build time and that TU is then linked into a larger program.
- Defer the choice of evaluation semantic until link time, load time, or runtime (prototyped in GCC): Instead of compiling with a fixed semantic, emit a function `__current_semantic` and branch on it. At link time, provide a definition of this function that deterministically returns the desired semantic. Link-time optimisation (LTO) may be used to reclaim the performance for unchecked builds; this has been shown to work reliably in the GCC prototype. Even without such reclamation, in practice the overhead of an ignored assertion can be reduced to two instructions and an always-taken branch.
- Further enhancements (not yet implemented): Instead of branching on a function call `__current_semantic`, we can branch on a static global variable set to a constant at link time. This improves the previous approach by providing trivial correct link-time selection; LTO is no longer needed because global variables are deduplicated by the linker.
- Represent the semantic in the ABI (prototyped for the Itanium ABI): A compiler can emit different symbols for the same function compiled with different semantics. The semantic (or the full configuration for how semantics will be chosen) can be directly incorporated into the mangled name, or we can use ABI tagging or symbol versioning. The linker can then deterministically choose a consistent version of the function at link time. This does *not* require any linker upgrades and can be accomplished on existing linkers by providing a linker map, which can be easily added to existing build systems such as CMake. A proof of concept for this approach has been shared with WG21. This strategy relies on contract assertion semantics being a language feature and does not work with preprocessor macros.
- Linker upgrades (future work): The ergonomics of the previous approach can be increased further by upgrading linkers to support deterministic choice of semantics directly, without the need to supply a linker map.

Concern 3: The impact of P2900 on dependency management is unclear

Summary

[P3849R0] suggests that the new build configurations introduced by contract assertions might complicate dependency management. The paper requests clarity on how these build configurations interact with real-world build systems or complex dependency graphs and asks what is expected of distributions that ship libraries as precompiled binaries.

Discussion Status

These topics were explored in [P3321R0] and discussed by SG15 in Wrocław, with no concerns raised by that group. No new information has been presented since.

Response

The different options for compiling contract assertions will be exposed through build flags. Distributions, package managers, and build systems already have existing methods to manage build flags, and users are familiar with their tradeoffs. P2900 introduces no new configuration dimension; rather, it replaces a proliferation of custom flags to control macro-based assertions with a single mechanism. It allows established practices to continue, integrates cleanly with existing infrastructure, and enables a variety of future implementation strategies to address many of the known use cases with different engineering tradeoffs, from optimised local builds to flexible shared binary distributions.

Details

P2900 provides a flexible model in which platforms can enable users to freely choose the evaluation semantics of any contract assertion. As discussed in Concern 2, compilers can implement a wide spectrum of conforming strategies with different tradeoffs, which will be exposed to the user via build flags.

Developers may understandably be concerned that adding another build flag for configuring contract-evaluation semantics will increase the already complex task of build configuration. Consider, however, that P2900 is intended to *replace* existing assertion mechanisms, typically one per library or framework, each of which introduces its own assertion macros and associated configuration flags. In addition, these mechanisms rely on the preprocessor and thus affect the token stream, making it harder for developers and tools to reason about the code. P2900 replaces this fragmented landscape with a single mechanism for controlling assertions that does not rely on the preprocessor. The result is an effective *reduction* in configuration complexity.

Importantly, contract assertions do not alter the existing C++ compilation model. Build flags that affect contract-evaluation semantics are no different in nature than any existing build flags that modify observable behaviour: `-ffast-math`, `-fno-exceptions`, `-ftrapv`, `-fwrapv`, flags that add sanitiser instrumentation, and so on. The exact shape and functionality of those build flags — including whether and how mixing different choices across translation units is supported — is, and remains, outside of the scope of the C++ Standard.

Dependency management in C++ routinely involves combining code compiled from source with code provided by multiple third parties as headers and precompiled object files. This also includes the system libraries provided to the user by their operating system. Ensuring ABI compatibility across all these dependencies is challenging enough; requiring consistent contract-evaluation semantics across all dependencies — including transitive ones unknown to the top-level program owner — is infeasible in most cases.

It follows that, with any viable assertion facility, the evaluation semantics will inevitably vary across dependencies. Users linking against a precompiled binary cannot generally know or control the set of flags used to build it — including those affecting the contract-evaluation semantics — and cannot produce a different configuration without access to the library source code. P2900 does not change this; libraries can still choose and document their preferred strategy, giving clients a useful range of options.

Crucially, contract assertions do not introduce new obligations on distributions. Vendors are no more obliged to ship binaries with checks enabled or to provide multiple binaries configured with different contract-evaluation semantics than they are today with macro-based assertions. Vendors remain free to choose what best serves their users: providing a single configuration (with assertions *ignored* for performance, *enforced* for debugging, or *quick-enforced* for maximum safety) or providing multiple configurations and letting clients choose which one to link against. The different hardening modes shipping with libc++ are a great example of using assertions in this way.

In practice, we do not expect build systems to face significant challenges in supporting contract-evaluation semantics. For example, Boost.Build already added such support on top of the available GCC and Clang implementations of P2900. Adding this support took less than an hour of implementation effort, demonstrating that contract assertions fit naturally into existing build models. The support includes documentation[1] covering scenarios such as static and dynamic linking against a library where both the library and `main()` are independently compiled with either *ignore* or *enforce*. By contrast, adding support for modules in the same build system has proven far more complex, requiring substantial re-engineering that is still ongoing.

Beyond the continued use of established practices, P2900 enables new capabilities for tooling (although none of them are *required* in order to use contract assertions effectively). As discussed in Concern 2, flexible-use binaries become possible by leveraging link-time or runtime selection of the evaluation semantic. In addition, `pre` and `post` sit at the boundary between caller and callee and can, therefore, be checked in the library, in the client, or both, independently from each other. Realising the full potential of such capabilities will require further work in implementations; a first implementation of caller-side checking is already available in GCC.

[1] Available at https://github.com/grafikrobot/cpp_contracts_example

Concern 4: P2900 violates the spirit of the ODR

Summary

[P3829R0], [P3851R0], and [ES-046] express concern that mixed-mode builds can create vulnerabilities that enable supply-chain attacks. [P3829R0] provides an example involving a function compiled with *quick-enforce* in one translation unit and *ignore* in another. When optimising the *quick-enforce* translation unit, GCC assumes the contract check is enforced and elides a caller-side null check. At link time, however, the *ignore* version of the function may be selected. [P3829R0] characterises this as a design flaw of P2900 and suggests that resolving it would require (1) making mixed mode an ODR violation, (2) significantly changing mid-level compiler intermediate representations, or (3) abandoning essential optimisations.

Discussion Status

EWG, in Hagenberg, considered the general concerns regarding mixed mode and rejected altering the ODR for contract assertions. The specific issue with interprocedural optimisation in GCC was discussed only on the reflector, where it was identified as a compiler bug unrelated to P2900. Recently, a reproducer for the underlying bug — which is unrelated to contract assertions — has been reported to GCC. No other new information has been presented since Hagenberg.

Response

The issue highlighted in [P3829R0] is a manifestation of a GCC bug unrelated to contract assertions (GCCBug121936). The suggestion that this bug is a flaw in P2900 appears to be a misunderstanding about the nature of the bug. Contract-evaluation semantics fit within the ODR in the same way as any other implementation-defined or unspecified (but still bounded) behaviour in the C++ language; the incorrect behaviour caused by the GCC bug is not conforming.

Details

The ODR does not require or guarantee consistent behaviour across all definitions of the same function ([basic.def.odr]); it requires only that their definitions consist of the same token sequences that resolve to the same entities. Behaviour fully specified by the C++ Standard or the ABI must be the same, but any implementation-defined or unspecified behaviour may vary across different compilations (or different evaluations) of the same code. Such variances can be observed, for example, due to differences in the evaluation order of function arguments and the precision of floating-point operations. Contract-evaluation semantics relate to the ODR in precisely the same way as those existing implementation-defined or unspecified behaviours.

It follows that if the compiler does not know exactly how the body of a function will be compiled, it cannot depend on any property of that function not fully specified by the C++ Standard or the ABI. *Derefinement bugs* ([Das16]) occur when a compiler performs interprocedural optimisations across the caller-callee boundary of an inline function in cases where the function is ultimately not inlined, thus violating this principle. Such optimisations inject invalid assumptions not supported by

what the ODR guarantees, resulting in a miscompiled program. Although [P3829R0] describes such optimisations on inline functions as 'desirable in the general case', they are fundamentally unsound.

Consider a function whose behaviour will depend on whether function arguments are evaluated left-to-right or right-to-left:

```
class Counter {
  int count = 0;
public:
  int next_count() { return ++count; }  // returns 1, 2, 3, ...
};

inline int one_or_minus_one() {
  Counter c;
  int result = std::minus()(c.next_count(), c.next_count());
    // unspecified argument evaluation order, result could be 1 or -1
  std::cout << result << '\n';
  return result;
}
```

Now, consider optimising a function that calls `one_or_minus_one` and then branches on the result:

```
void optimisable() {
  if (one_or_minus_one() > 0)
    /*... branch body ...*/;
}
```

A compiler may legitimately optimise `optimisable` in several ways:

- Inline the body of `one_or_minus_one`, always print `-1`, then return.
- Inline the body of `one_or_minus_one`, always print `1`, then execute the branch body.
- Make a weakly linked call to `one_or_minus_one`, then test the result and conditionally execute the branch body.

There are, however, options that are not legitimate for a compiler to choose:

- Make a weakly linked call to `one_or_minus_one`, then assume that it always returns `-1` and return (skipping the branch and its body).
- Make a weakly linked call to `one_or_minus_one`, then assume that it always returns `1` and execute the branch body.

In these last two cases, if the TU that contains the selected definition of `one_or_minus_one` differs in its choice of argument evaluation order from the TU that contains `optimisable`, the linked program will exhibit nonsensical behaviour when `optimisable` is invoked:

- Print `1` and then return without executing the branch body.
- Print `-1` and then execute the branch body.

Neither of these two behaviours represents a conforming execution, and thus each is an example of a derefinement bug — directly analogous to the case described in [P3829R0].

Because such optimisations are unsound, both Clang (LLVMPR26774) and GCC (GCCBug70018) disabled them nearly a decade ago. The behaviour reported in [P3829R0] is a regression of the same issue in GCC 14, entirely unrelated to contract assertions. During upstreaming of GCC's P2900 implementation, we filed a complete bug report and reproducer in GCCBug121936.

The suggestion in [P3829R0] that avoiding such bugs requires a redesigned intermediate representation is inaccurate. Existing intermediate representations are sufficient to represent contract assertions with specific or dynamic semantics if compilers refrain from assuming that other translation units will produce the same intermediate representation and process all unspecified aspects of its behavior exactly the same way.

Concerns about harming performance by disabling these optimisations are equally unfounded. The optimisations are suppressed only when an inline function is *not* inlined and is instead invoked indirectly, which already incurs overhead that likely dwarfs any benefits from such optimisations. Clang made this tradeoff long ago without user complaints.

For any preprocessor-based assertion facility, mixing modes in inline functions would result in clear ODR violations; different tokens are being passed to the compiler with no indication that they represent the same assertion, and the compiler is then asked to optimise across them. Such ODR violations inevitably lead to the kind of security issues that [P3829R0] raises. By contrast, the evaluation-semantic model in P2900 does not suffer from these issues; it provides both implementation and user flexibility while putting a reasonable onus on the compiler to guarantee one of a specific set of defined behaviours — the evaluation semantics an assertion has been compiled with — in all cases.

Concern 5: P2900 does not work well with modules

Summary

[P3835R0] raises the question of how modules relate to contract assertions and whether they could address the various concerns regarding the configuration of contract-evaluation semantics.

Discussion Status

The interaction between P2900 and modules was previously raised in [P3573R0], responded to in [P3591R0], and further discussed in EWG in Hagenberg. No new information has been presented since.

Response

Modules are not a solution to every concern, but they do provide capabilities unavailable in a purely header-based model and, therefore, represent a distinct avenue for configuring contract-evaluation semantics.

Details

In principle, inline functions in a BMI could carry additional information, such as contract-evaluation semantics. Doing so would give implementations more options for resolving inconsistent build flags (see Concern 3). However, because a BMI can itself be compiled multiple times with different flags in the same build (often matching the flags used by the importer), modules remain only a partial solution to the broader problem of ensuring flag consistency across object files. That problem is not specific to contract assertions but applies equally to other forms of implementation configuration.

More generally, no single right answer dictates how compilers should prioritise conflicting configuration information across module boundaries. Warning flags illustrate this difficulty: should a compiler rely on the flags of the importee or of the importer when instantiating a templated function?

The same question arises for contract evaluation semantics. Users will reasonably expect control over this decision, and modules at least provide one natural approach to enabling such control, unlike header-based definitions, which lack ownership information, or link-time decisions, which require more involved implementation efforts (see Concern 2). All users of modules, however, will not want to base the decision solely on the module to which a contract assertion belongs since many builds will control checking of the functions a module uses, not just those that belong to a module.

Concern 6: Too much in P2900 is implementation-defined

Summary

[US 26-051] and [FR-005-054] raise the concern that P2900 contains 'too much implementation-defined behavior'. [RO 2-056] states that P2900 'relies excessively on implementation-defined behavior'. [P3829R0] states that 'almost everything in P2900 is implementation-defined'.

Discussion Status

This concern was raised in [P3573R0], with a response in [P3591R0]. Both papers were discussed in EWG in Hagenberg. [P3321R0] lists the various options available to implementations for the implementation-defined parts of P2900. These papers were discussed by SG15 in St. Louis and Wrocław with no concerns raised by that group. No new information has been presented since.

Response

C++ has always made behaviours implementation-defined when they are, by necessity, platform-dependent or when not all programs on a platform will want the same choice of behaviour. P2900 is no different in the set of behaviours it specifies as implementation-defined. In practice, the only new implementation-defined behaviour that will require regular attention from the user is the selection of contract-evaluation semantics. None of these implementation-defined behaviours alter the way contract assertions are written nor do any represent an unresolved design gap.

Details

A detailed response has already been provided in [P3591R0] (see page 8) and discussed by EWG in Hagenberg. In St Louis, SG15 discussed [P3267R1], which explores many ways a compiler might approach the implementation-defined parts of contract-assertion evaluation. In Wrocław, SG15 discussed [P3321R0], which discusses the full list of implementation-defined behaviours from a general tooling perspective. Both papers received a positive response, and SG15 explicitly stated there were no tooling concerns to resolve.

P2900 introduces exactly five implementation-defined properties:

1. The specific termination mode used by *enforce* and *quick-enforce*
2. The exact behaviour of the default contract-violation handler
3. Whether the contract-violation handler is replaceable
4. Choice of evaluation semantic for each contract-assertion evaluation (configuration mechanism and default)
5. The maximum number of repeated evaluations of a contract assertion

The first three items reflect platform diversity. The most appropriate mode of termination in response to a diagnosed bug differs significantly across environments, with distinct tradeoffs. Diagnostic output mode and formatting are equally variable. The replaceability of the violation handler is viewed by some platforms as a necessary feature for even the most basic viability of the proposal and by other platforms as a security risk.

The remaining two items are controlled by compiler settings. The choice of evaluation semantic will typically be exposed through compiler flags (see Concern 3), whose mechanics the C++ Standard has never prescribed. Finally, the maximum number of repeated evaluations may depend on whether the compiler supports caller-side checking ([P2751R0], [P2780R0], [P3264R1], [P3267R1]) and the ABI; the only requirement is that this number have a known upper bound ([P3119R1]).

Crucially, none of these implementation-defined properties alter the way contract assertions are written, nor do any of them represent an unresolved design gap. Instead, these properties are precisely those aspects where no single solution can serve all platforms and users. Far from being a weakness, this flexibility is what ensures that contract assertions are usable across the full breadth of C++ implementations.

Of these five implementation-defined properties, only the choice of evaluation semantic will need regular attention by users since they will typically be using different evaluation semantics depending on their goal. The other four properties are typically defined by the platform or configured once for a particular project and need not be touched again unless the requirements change.

The following overview table lists the choices made for each implementation-defined property in the current implementations of P2900 in GCC and Clang. Note that these choices pertain to the respective development branches; the behaviour, flag name, and exact message format eventually accepted by the community after upstreaming may differ.

	GCC	Clang
Termination mode on contract violation	`std::terminate()` (will add a flag to allow the user to select trap or `std::abort()` instead)	`std::terminate()` on *enforced* contract violations, a trap instruction on *quick-enforced* violations
Behaviour of default contract-violation handler	Print message: `contract violation in function ... at ... assertion_kind: ... semantic: ... mode: ... terminating: ...`	Print message: `<file>:<line_number>: <comment> failed`
Replaceability of contract-violation handler	Replaceable on all platforms that support overriding weak definitions	Replaceable on all platforms that support overriding weak definitions
Configuration mechanism for evaluation semantics	Per TU with build flag `-fcontract-evaluation-semantic=...`	Per TU with build flag `-fcontract-evaluation-semantic=...`
Default evaluation semantic	*enforce*	*enforce*
Maximum number of repeated evaluations	Two, if caller-side checking is enabled; one otherwise	One (will become two when caller-side checking is eventually implemented)

Concern 7: P2900 relies on guidelines the compiler cannot check

Summary

[P3849R0] states that the primary requirement for contract assertions is that a contract check 'must not change the behaviour of a program' and then suggests that requirement is unlikely to be met in practice, concluding that a feature that 'requires extra directives to use correctly' should not be added to C++26.

Discussion Status

The desire to produce rules that a compiler can enforce to restrict predicates to only those that are nonproblematic has been put forth in papers such as [P2680R1], [P3285R0], and [P3362R0]. These papers have been given ample committee time and not achieved consensus. No new information has been presented since.

Response

Avoiding modifications to program state in contract predicates is standard practice and is generally a good starting point. The actual requirement for viable predicates, however, is that they must not change the *correctness* of a program (Section 3.1.1 of [P2900R14]), i.e., that its predicate not be *destructive*. This requirement is not specific to P2900 and applies to any assertion facility including C `assert`. Writing nondestructive contract predicates is generally undemanding for users, but identifying them based on language rules without making any nontrivial predicates ill-formed is effectively impossible.

Details

This concern rests on two misunderstandings. The first is the suggestion that the compiler could somehow enforce that a contract check has no side effects. This idea, along with the suggestion that contract-assertion predicates should be written in a new sublanguage free from undefined behaviour, has been raised repeatedly ([P2680R1], [P3285R0], [P3362R0]). Despite years of discussing these ideas in SG21, SG23, and EWG, no proposal capable of achieving this goal has ever been produced, and no consensus has been achieved in these groups for pursuing this direction. The evidence strongly suggests that enforcing such constraints is impossible without making all but the most primitive C++ expressions inside a contract predicate ill-formed ([P3499R1]), rendering the feature unusable in most real-world scenarios.

The second misunderstanding is that compile-time enforcement of side effects, even if it were achievable, would be both necessary and sufficient to prevent bad predicates. This misunderstanding is rooted in a common conflation of two distinct concepts:

- *Destructive side effects*, which alter program correctness, i.e., whether the execution of a program conforms to its specification ([P2712R0], Section 3.1.2 of [P2900R14])
- *Observable side effects*, which modify program state and/or alter observable behaviour

The difference between these two side effect varieties is crucial to understanding whether a side effect constitutes a material problem. For example, logging to cout during a contract check is observable but is destructive only if the specification makes the output of cout significant for the program's correctness. The same applies to contract checks that allocate memory or temporarily lock a mutex. Conversely, a contract check verifying that an array is sorted may have no observable side effects inside the C++ abstract machine ([intro.execution]/7) but is destructive if it violates specified complexity guarantees. Beyond observability in the abstract machine sense, even the simplest contract check might violate performance requirements that are more stringent than complexity guarantees (e.g., in a real-time environment) because the check inserts instructions that will take nonzero time to execute.

The fundamental requirement for writing contract assertions is that they not be *destructive* — a property that, fundamentally, is not objectively verifiable by a compiler but requires a human who understands the plain-language specification and the particular context in which the contract assertion might be checked. This requirement is not specific to P2900's design; it applies to any correct usage of an assertion facility in any programming language, including existing C++ facilities such as C `assert` and other preprocessor-based assertion macros.

Decades of experience with these facilities have shown that destructive side effects from predicates are easily identified during development and testing and are rarely an issue. We have been teaching for a long time that assertions should not have side effects (e.g., Rule 68 of [Sutter04]), and users have learned to use them correctly and effectively. P2900 strengthens this further with `const`-ification (Section 3.4.2 of [P2900R14]), which reduces the likelihood that a destructive side effect could be written unintentionally without causing a compiler error.

Concern 8: `const`-ification is problematic

Summary

[P3851R0] and [ES-055] raise concerns about `const`-ification for the teachability and understandability of code. [CZ 4-058] observes that `const`-ification complicates automatic assertion insertion by tooling. Additionally, [P3851R0] and [ES-073] state that a parameter used in a postcondition assertion 'will magically become `const`'.

Discussion Status

Objections to `const`-ification were raised previously in [P3478R0], [P3506R0], and [P3573R0]. A detailed analysis of alternative approaches and their costs and benefits was provided in [P3261R2]. This issue was discussed extensively in both SG21 and EWG. In Wrocław, EWG reached *consensus against* removing `const`-ification, which was reaffirmed in Hagenberg. The concern in [CZ 4-058] that `const`-ification could increase the difficulty of automatic assertion insertion by tooling and that `const`-ification could be replaced with erroneous behaviour are new. Otherwise, no new information has been presented since Wrocław.

Response

`const`-ification does not change the `const`-ness of a parameter used in a postcondition assertion. It can change overload resolution in contract-assertion predicates, but this change impacts only code that is already unusable in practice ([P3261R2]). No compelling real-world examples of correct assertions rendered incorrect by `const`-ification have been produced. By contrast, when existing implementations of P2900 in GCC and Clang were applied as a replacement for legacy assertion libraries, `const`-ification revealed genuine bugs in existing libraries.

Details

[P3071R1] added `const`-ification to the proposal to guard against inadvertent modifications of program state in assertion predicates while still allowing such modifications when explicitly needed. The poll to incorporate this feature into P2900 had strong consensus.

As P2900 progressed through EWG, opposition to `const`-ification was raised ([P3478R0], [P3506R0], [P3573R0]) and centred on it potentially altering overload resolution: a member function called in a contract-assertion predicate will select the `const` overload, yet the non-`const` overload would be selected outside a predicate. In response, [P3261R2] presented a comprehensive analysis of possible approaches, addressing in detail the concerns now reiterated in [P3851R0]. Following extensive discussion, both SG21 and EWG reaffirmed consensus in favour of `const`-ification.

Section 2.3.1 of [P3261R2] provides a flow chart demonstrating that circumstances where `const`-ification introduces friction are exceedingly rare. Supporting this line of reasoning, applying `const`-ification directly to several large codebases ([P3268R0], [P3336R0]) produced no reported cases of breakage and revealed genuine bugs in those codebases. No compelling realistic code example has been produced where `const`-ification introduces a bug.

Concerns about the teachability of `const`-ification were addressed in Section 4.1 of [P3261R2]. This analysis concludes that omitting `const`-ification would encourage na ive users to write broken assertions, and including it prevents common classes of user errors. Others have also since noted that the mechanism is not difficult to teach and not novel: C++ already treats member access expressions in member functions as implicitly `const` in exactly the same way, even applying the same transformation (with no direct use of the `const` keyword) to by-value captures in lambda expressions.

The novice user of contract assertions needs to be taught just two things, neither of which involves even knowing about the existence of `const`-ification: (1) express what is always true in the assertion's predicate and (2) avoid all side effects. The expert user might need to learn slightly more if they encounter `const`-challenged code or a compilation failure that prevented them from making a mistake, but that new lesson is similar to understanding the implications of `const` on member functions and lambdas.

The use case mentioned by [CZ 4-058], automatically inserting assertions that would willingly risk modifying existing objects by applying a `const_cast` nested within the predicate, has not been considered. Such tooling might be useful, but language specification should prioritise the human ability to effectively write correct code over the ease of implementing such tools. More notably, without careful inspection of the expressions being automatically placed in contract assertions to (somehow) be sure they will not be destructive it is vastly more likely that such tooling would be introducing bugs that could easily have been prevented by having `const`-ification and not attempting to automatically subvert it with `const_cast`.

The suggested fixes for `const`-ification in [ES-055] and [CZ 4-058] do not address whether the result name should be treated consistently with other variables that currently undergo `const`-ification. This inconsistency was a major concern for a number of participants before `const`-ification was adopted. The proposed change in [CZ 4-058] to treat modifications as erroneous behaviour was not explicitly considered by [P3261R2] but exhibits similar drawbacks to many of the alternatives discussed there: it lacks an existing escape hatch, provides no clear bounds on what it applies to, and converts what would be a compile-time failure into a runtime failure.

[P3851R0] and [ES-055] inaccurately observe that a parameter used in a postcondition assertion 'will magically become `const`'. The described behaviour appears to have been confused with the requirement that using a non-reference parameter in a postcondition assertion requires that parameter to be explicitly declared `const` by the user. The clamp example in Section 3.4.4 of [P2900R14] should make clear why this requirement is needed, as do the issues with adding postcondition assertions to coroutines ([P2957R2]). Post-C++26 extensions, such as postcondition captures ([P3098R1]), will provide more flexible support for using non-`const` non-reference parameters in postcondition assertions.

Concern 9: Global contract-violation handlers are problematic

Summary

[P3829R0] suggests that global handlers, such as the contract-violation handler in P2900, are difficult to use in practice — citing experience with `std::unexpected`[2] — and argues for specifying different handlers for different classes of bugs.

Discussion Status

The global contract-violation handler was adopted into P2900 when SG21 approved [P2811R7], and it had consensus in EWG. Local violation handlers have been proposed in [P3400R1] as a post-C++26 extension. No new information has been presented since.

Response

The global contract-violation handler is a central feature of P2900 and one of the primary motivations for pursuing standardisation of contract assertions. It allows the application owner to decide how to report and manage violations when integrating libraries from a wide variety of third-party suppliers, which is not possible with nonstandard solutions.

Details

The idea to specify different handlers for different assertions is not new; it was explored in early proposals, found to scale poorly, and explicitly abandoned in favour of a global handler (see Section 3.5.9 of [P2899R1] for history and rationale). Crucially, a single, uniformly replaceable, user-definable contract-violation handler across the entire application, including third-party libraries, is essential when assertions are employed at scale because it allows the application owner to decide what happens when a contract violation occurs at every level of the program stack without having to be aware of nested implementation details. Providing this functionality is a primary motivating use for pursuing a standard contract-assertion facility.

The removal of `std::unexpected` in C++17 should not be misread as evidence against global handlers. It was removed as part of eliminating dynamic exception specifications ([N3051], [P0003R5]), not because the concept of a global handler was inherently flawed. The current consensus is clear: global handlers can be problematic when used for error handling and recovery but are effective for logging, diagnostics, and guided shutdown when program bugs are detected (see, for example, [Sutter19]). This latter behaviour is exactly the role of P2900's contract-violation handler.[3]

C++ already includes several global handlers for this purpose (e.g., `std::set_new_handler`, `std::set_terminate`, signal handlers), and similar mechanisms are widely and successfully used in

[2] This example refers to the callback function `std::unexpected` in header `<exception>` that was deprecated in C++11 and removed in C++17, not the class template in header `<expected>` introduced in C++23.

[3] As a niche use case, the contract-violation handler may also throw an exception. Doing so allows for a recovery attempt when the exception facility is able to facilitate that recovery, whereas `std::unexpected` is invoked when exception handling has failed.

major frameworks such as Qt and in game engines. Unlike `std::unexpected`, these facilities are not intended for recovery, and experience with them has been positive.

For C++26, requiring users to specify a per-assertion handler is deliberately a nongoal: the advantages of controlling how errors are handled by many unrelated third-party libraries is a primary motivating factor of having contract assertions as a language feature. Post C++26, opting into local overrides or additions to that global handler might be possible through adoption of a proposal such as [P3400R1], but such a local handler will always be an opt-in that requires additional syntax that extends the initial facility in C++26.

Concern 10: Observing consecutive contract assertions is dangerous

Summary

[P3851R0] and [ES-047] raise the concern that multiple contract assertions, when *observed*, might result in undefined behaviour because an earlier contract assertion might serve as a precondition check for being able to evaluate the predicate of a later assertion, and they characterise this possibility as a safety issue.

Discussion Status

This concern has surfaced regularly any time the *observe* semantic was discussed, e.g., when *observe* was reintroduced by [P2877R0] and again when [P3582R0] proposed a mitigation similar to that in [P3851R0]. After discussion of [P3582R0] in SG21, no one — *including the author of that paper* — was in favour of pursuing such a mitigation. No new information has been presented since.

Response

The *observe* semantic is an indispensable tool when introducing new contract assertions into existing code, but continuing past a failed assertion always comes with a risk. The idiomatic solution is to combine dependent predicates into a single assertion, thus avoiding the risk of evaluating the second condition after the first fails. Proposals to automatically skip subsequent assertions after an *observed* contract violation are problematic because doing so could suppress an unrelated *enforced* check and result in a worse outcome.

Details

In general, the *observe* semantic is an essential tool when introducing new contract assertions into deployed code without risking unexpected termination. By definition, *observe* carries the risk that an assertion that would have caught a bug when *enforced* instead allows execution to continue, but it does notify the violation handler; the *ignore* semantic would leave everyone ignorant of the problem. Therefore, *observe* should be understood to improve one's knowledge of bugs in a system, but just like *ignore*, it is unable to prevent the bugs from having an impact and in rare cases expedite the effects of a bug.

The canonical example of this concern, as alluded to in [P3851R0], involves performing a null pointer check in one precondition assertion and then dereferencing the same pointer in the following precondition:

```
void runProgram(Program* p)
  pre(p != nullptr)
  pre(p->is_runnable());
```

A call to the `runProgram` function above with a `nullptr` value and assertions observed will invoke the contract-violation handler, notify the user of a bug, and then have undefined behaviour when dereferencing the null pointer as part of evaluating the second precondition. Note that the compiler is not allowed to use this undefined behaviour to optimise out the first precondition assertion and the associated call to the contract-violation handler, since P2900 specifies an *observable checkpoint* ([P1494R5]) after the contract-violation handler returns.

An easy, convenient, and readable way to obviate this concern is to combine the dependent predicates into a single assertion, taking advantage of short-circuit evaluation with the `&&` operator:

```
void runProgram(Program* p)
  pre(p && p->is_runnable());
```

Attempting to instead address this concern by automatically skipping later checks after the first observed check fails is unsound. In functions with orthogonal preconditions, the suppressed check might hide more critical failures than the earlier check that failed. If the earlier check is being observed while the more important check was enforced, this mismatch can lead to a complete failure of enforced checks preventing entry into the following code after a violation:

```
void launchMissiles(int numMissiles, Target target)
  pre(numMissiles > 0)          // added recently, should be observed
  pre(target != Target::self);  // well established, should be enforced
```

Skipping the second check if the first fails would allow a potentially dangerous call, `launchMissiles(0, Target::self)`, to proceed into the body of the function. Perhaps this unintended result will not be a problem with 0 missiles, or perhaps the body of the function depends more concretely on the precondition and assumes at least one missile is always available to launch. Importantly, when adding a new check to an existing function, UB introduced in the contract assertion itself would most likely have inevitably occurred directly in the implementation of the function to which that contract assertion was added.

When dependent checks cannot be written as combined predicates, the flexible model of P2900 — e.g., every evaluation of a contract assertion can have a different semantic, the choice of which is implementation-defined — provides the necessary implementation freedom to realise more specific mitigation strategies. For example, a build mode in which assertions following an *observed* violation are categorised through static analysis as related (perhaps when one makes use of the same variables whose values were tested in an earlier assertion) are *ignored* is conforming under P2900, and users who prefer such behaviour can solicit this option from their toolchain vendors.

Concern 11: Treating exceptions as contract violations is infeasible

Summary

[FI-071] comments that no implementation or deployment experience of P2900 exists for non-Itanium ABIs and adds that Microsoft considers it infeasible to treat exceptions thrown from the evaluation of contract predicates as contract violations.

Discussion Status

The same position from Microsoft was raised previously in [P3506R0], addressed in [P3591R0], and given due consideration by EWG in Hagenberg. No new information has been presented since.

Response

The rationale for preventing exceptions from arbitrarily escaping the evaluation of a contract-assertion predicate have been thoroughly established, beginning with [P2751R1]. Prioritising correctness over hypothetical performance costs is essential for P2900's design. Moreover, no evidence suggests the performance cost is significant. The overwhelming majority of predicates are trivially non-throwing, and for all such predicates there is no cost in binary size or runtime overhead on any platform.

Details

The process of determining how to handle the various ways in which contract-assertion evaluation might complete ([P2751R1]) made clear that exceptions must be treated carefully. Some participants argued that exceptions should never escape such evaluation, since doing so would introduce new control-flow paths when enabling assertion checking — an approach that caused issues in the C++20 Contracts proposal ([P0542R5]). Others required the ability to recover from certain exceptions, such as `bad_alloc`, regardless of whether they originated from a contract-assertion evaluation.

The approach in P2900 is the only known solution that satisfies both groups: exceptions thrown during predicate evaluation are treated as contract violations and passed to the violation handler, allowing user-defined recovery strategies while maintaining sound control-flow semantics.

Concerns about the cost of this approach in binary size and run time were raised in [P3506R0] and again in [P3573R0] and discussed in [P3591R0], which elaborated on the reasons why there will be no overhead for predicates that are not potentially throwing, which in our experience is the vast majority of assertion predicates.

Both SG21 and EWG concluded that prioritising the speculative overhead of a small subset of predicates over ensuring that contract assertions have well-defined, analysable control flow would be unsound. Passing exceptions to the contract-violation handler gives users the ability to reason about their program and the necessary control to choose the strategy most suitable for their use case.

Concern 12: P2900 does not support static analysis

Summary

[FI-071] expresses the concern that P2900 does not provide 'serious support' for static analysis tools.

Discussion Status

These concerns were raised in [P3362R0] and responded to in [P3376R0] and [P3386R1]. All three papers were discussed by EWG in Wrocław. No new information has been presented since.

Response

The concern in [FI-071] that P2900 might not be a good fit for static analysis tools is unsupported by evidence. P2900 directly addresses the limitations of macro-based assertions that static analysis tools already exploit today, and the vendors of these tools have consistently confirmed that its design will substantially improve their ability to detect bugs. Some static analysis providers (such as CodeQL) are already actively pursuing support for P2900 contract assertions in their tools.

Details

During the development of P2900, we worked closely with multiple vendors of static analysis tools (JetBrains, Synopsis, PVS-Studio, CodeQL). These tools already exploit macro-based assertions, such as C `assert`, for flow analysis and range analysis to detect bugs ([P3386R1]).

However, the usefulness of macro-based assertions for static analysis is limited by three factors. The first is the lack of a uniform syntax; projects often use custom macros that the tool cannot recognise without specific configuration or at all. The second is that macros may be removed from the token stream by the preprocessor, rendering them invisible to the tool. Finally, the third is their placement inside function bodies, which are much harder to analyse than function declarations and limit single-TU analysis. This last limitation in particular severely limits the ability of static analysis tools to consider assertions at scale.

These limitations have led to bespoke mechanisms to annotate preconditions and postconditions on function declarations, including embedded markup in comments, custom attributes, or external files. For example, [SAL] uses tool-specific attributes that can be added to certain declarations, and [CodeQL] inspects comments and some vendor-specific assertion macros and can extract inferred contract assertions to external files.

Contract assertions as specified in P2900 remove all three limitations. They provide a portable, *standard* syntax that will be understood by all tools; they are not preprocessor macros; and they can be placed on function *declarations*, obviating the need to analyse function bodies or add tool-specific annotations to reason about code correctness across function call boundaries.

During the development of P2900, static analysis vendors consistently confirmed that these features will significantly improve the ability of their tools to find bugs once contract assertions are adopted

and used. Most recently, [Martin25] demonstrated how P2900 can enable static proofs beyond conventional flow and range analysis. They combine the CodeQL static analyser with the Z3 constraint solver to validate a wide range of contracts and present a case study evaluating the effectiveness of these techniques on real-world codebases.

Concern 13: P2900 is too complex

Summary

[P3829R0] suggests that the design of P2900 is 'too complex' and that the value it adds may not justify the cost in added language complexity.

Discussion Status

The concern regarding complexity in [P3829R0] mirrors that of [P3573R0], which was discussed by EWG in Hagenberg. No new information has been presented since.

Response

By dint of how quickly all its essential parts can be explained and how little effort has been required for production-quality implementations in two compilers, P2900 is clearly simpler than many recent major language features. The real complexity lies in carefully analysing the problem space and constraints, not in the feature's design, implementation, or user experience. The broad, sustained effort over more than two decades to bring contract assertions to ISO C++ clearly signals industry demand for this feature; the strong plenary consensus in Hagenberg to include P2900 in the C++26 working draft is further evidence that its value is worth the cost.

Details

Complexity is hard to measure objectively, as illustrated by the 'deeply unscientific informal polls' in [P3829R0] with 86 respondents — 79% of whom do not use C++. Note also the apparent tension in [P3573R0] and [P3829R0], which characterise P2900 as simultaneously too complex and insufficiently feature rich. Moreover, discussion volume is often mistaken for design complexity: that contract assertions have been actively debated for many years is evidence of the scrutiny WG21 has given it, not of the design complexity or of difficulty of use.

The real measure of complexity in C++ is programmer experience, especially for simple tasks. From this perspective, P2900 is straightforward: everything needed to begin using the feature effectively can be explained in minutes ([Doumler24], [Nash25]). This ease of use stands in sharp contrast to other major features added in C++26, such as executors ([P2300R10]) or reflection ([P2996R13]).

Implementation experience reinforces this point. *Complete* implementations of P2900 in GCC and Clang were produced relatively quickly by a tiny team ([P3460R0]) and with limited impact on those compilers. The implementers reported that P2900 is orders of magnitude simpler to support than modules, concepts, reflection, or even lambdas.

Finally, the length of P2900's specification reflects not intrinsic complexity, but the need to integrate contract assertions precisely with the intricacies of the existing language: trial constant evaluation ([expr.const]/28), `const` parameters in coroutines that are not truly `const` ([dcl.fct.def.coroutine]/14), special rules for trivially copyable objects passed in registers ([class.temporary]/3), and so forth. P2900's specification ensures that such edge cases yield predictable outcomes the user can reason about — usually a compiler error — rather than undefined behaviour. The detailed specification is evidence of careful, rigorous integration into the existing language — something earlier attempts, such as [P0542R5], failed to achieve.

Thus, the true complexity of P2900 lies in the intellectual effort to understand the entire problem space and the constraints on any solution that would address such a diverse set of use cases consistently, coherently, and as simply as possible. Even the first step into that problem space — understanding how we can identify that a particular behaviour needs to be well-defined but also treated as incorrect — has just begun to be absorbed by WG21 in this standardization cycle with the adoption of erroneous behaviour. The significant investment in P2900's design, specification, standardisation, and implementation in two major compilers, involving many contributors across multiple organisations, clearly demonstrates that the value it brings to the C++ ecosystem far outweighs any alleged cost in increasing language complexity.

Concern 14: P2900 lacks important features

Summary

[P3829R0] states that P2900 'lacks important features' and lists per-subsystem semantics, per-severity reporting, nonglobal contract-violation handlers, concise syntax for reusable checks, and user-defined messages as examples. [P3835R0] suggests in-source control of evaluation semantics is necessary because Clang and libc++ have needed that functionality in experimental ports of their existing hardening macros to P2900 contract assertions. [P3851R0], [ES-072], and [ES-074] suggest that support for `pre` and `post` on virtual functions and function pointers are necessary.

Discussion Status

SG21 and EWG iteratively selected among the many potential features that should be included in P2900. All the requested features have been discussed in various papers; no proposals that included them gained consensus in EWG. Most recently, [P3573R0] brought up many of these options; during discussion in EWG in Hagenberg, many of the features (being requested again) were individually polled for inclusion, and no consensus to add them to P2900 was reached. No new information has been presented since.

Response

P2900 provides a minimal, coherent core facility that supports real-world use cases and leaves room for evolution. The requested additional features can naturally extend P2900; most already have active proposals targeting C++29 and are being explored in experimental compiler extensions. This incremental approach follows the proven pattern of C++ features, where a minimal usable core is standardised first and more advanced functionality added later. Delaying standardisation until all conceivable use cases can be supported would block adoption and deprive the community of the immediate value contract assertions provide.

Details

The real-world use cases for contract assertions have been extensively studied ([P1995R1]) and are addressed by P2900 and its proposed extensions. The lifetime of P2900 really began when SG21 agreed to follow the path in [P2695R1] to pursue a specific plan to gain consensus on a minimal viable product (MVP) for contracts in C++. Starting with an MVP enables us to provide an already-standardised foundation on which consensus can be built for higher-level features that come later. A significant part of the discussion at every stage of P2900's development was about how to settle the most fundamental decisions about contracts so that future evolution was never disregarded even when it was not prioritised as part of the MVP itself.

All the requested features have been previously discussed in meetings, reflector messages, and papers.

- The lack of syntactic grouping of assertions — which provides per-subsystem semantics and per-severity reporting — was raised in [P3573R0] and answered in detail on page 9 of [P3591R0]. These capabilities are being proposed in [P3400R1] as a post-C++26 extension.
- User-defined messages are likewise proposed in [P3099R0] as a post-C++26 extension.
- Less commonly requested extensions, such as syntax for reusable checks, do not yet exist in the form of fully specified proposals but have been outlined in [P2755R1] and [P2885R3].

All these features are pure extensions of P2900, and the design explicitly anticipates them without requiring modifications or ABI breaks.

[P3851R0] and [ES-074] request supporting `pre` and `post` directly on function pointers, repeating identical requests in previous papers ([P3173R0], [P3506R0]). The design space for such a feature was extensively explored in [P3327R0] but found to be infeasible. EWG reviewed and confirmed this conclusion in St. Louis. Instead, a novel feature would have to provide equivalent functionality. A promising direction are f*unction types with usage*, proposed in [P3271R1]; an alternative design direction has been proposed in [P3583R0]. This design space is actively being explored; a detailed overview and history can be found in Section 3.3.6 of [P2899R1].

[P3851R0] and [ES-072] further request supporting `pre` and `post` on virtual functions, again repeating identical requests in previous papers. The history of this feature's adoption by EWG in St. Louis and subsequent removal in Hagenberg is described in Section 3.3.2 of [P2899R1]. Unlike the other suggested extensions to P2900, `pre` and `post` on virtual functions do have a proposal ([P3097R0]) that is fully specified, has been reviewed and approved with strong consensus in EWG, has been

reviewed by CWG, has been implemented in GCC, and could be re-added to the C++ working draft any time EWG wishes to do so.

[P3835R0] suggests that contract assertions are unusable without a mechanism to allow semantics to be tied to components. An actual need is a way to produce control over semantics similar to those available through having a family of different macros that are flexibly controllable, such as the various hardening levels provided by libc++ or the assertion levels provided by BDE. Clang has experimentally provided this using attributes because this functionality is needed to provide comparable levels of features to those libc++ already has in its hardening macros, which predate P2900. That P2900 itself does not yet provide that level of in-source flexibility is well known, will be addressed by the capabilities in [P3400R1], and is not a hindrance to those seeking to adopt contract assertions for other purposes.

Incremental standardisation has repeatedly proven effective in C++: early versions of constexpr functions allowed only a single return statement, and return type deduction was initially limited to lambdas. Standardising a minimal but usable core enables adoption and real-world experience while leaving room for vendor experimentation that allows innovation and informs evolution.

Many users of C++ will benefit immediately from contract assertions (as soon as they are using a compiler release that supports them) through the improvements to libc++'s hardening macros that leverage `contract_assert` along with compiler extensions to meet their more involved deployment needs. Even before that, standard library macros that conditionally support the use of contract assertions will benefit from added attention even on older platforms where contract assertions are not yet available. Furthermore, the feature set of P2900 is sufficient for many users to gain initial benefits from using contract assertions directly in their code. Broader extensions, such as [P3400R1], will widen this pool, with future evolution driven by the real-world deployment that P2900 allows.

By contrast, delaying standardisation to add features blocks adoption, deployment in real-world codebases, and evolution. Most importantly, it deprives the C++ community of the benefits contract assertions are designed to immediately deliver: identifying bugs, improving program correctness, and strengthening functional safety — incrementally, portably, and scalably.

Concern 15: P2900 makes future desirable features harder to add

Summary

[P3829R0] raises concerns about interactions between P2900 and hypothetical future C++ proposals.

Discussion Status

Integration with future features was discussed extensively during P2900's development. Requirements were analysed in [P2885R3]; the possibility of deep `const` was examined in [P3261R2] and rejected via poll in both SG21 and EWG. No new information has been presented since.

Response

P2900 has been explicitly designed with consideration of features that might be added. Nevertheless, the proposal, *following WG21 practice*, avoids waiting for facilities that have not yet proven sufficiently compelling or feasible to pursue their standardisation, such as deep `const` or lazy function argument evaluation.

Details

Procedurally, delaying the standardisation of P2900 due the concerns of [P3829R0] about interactions with hypothetical proposals would be against WG21 practice. According to [SD4], 'we do not significantly delay progress on concrete proposals in order to wait for alternative proposals we might get in the future'. The interactions described in [P3829R0] are speculative and do not reveal design flaws in P2900.

First, [P3829R0] suggests that P2900 could complicate the standardisation of deep `const`. The paper asks whether precondition assertions would require modification to take a deep-`const` view of arguments or whether new core-language keywords would be needed.

We agree with the authors of [P3829R0] that having deep `const` in the language might be beneficial. Yet in more than four decades of C++ evolution, no proposal for deep `const` has ever been brought forward, and it appears doubtful that one will ever materialise. Properly specifying deep `const` is a daunting undertaking since it would require WG21 to resolve numerous long-standing, intricate technical questions (which [P3829R0] partially acknowledges and [P3261R2] explored more deeply when considering alternate designs for `const`-ification) before any interaction with contract assertions could be considered.

Second, [P3829R0] suggests that P2900 could interfere with a hypothetical proposal for generic decorators — a mechanism to encapsulate code that runs before and after function calls. While decorators have their use cases, no actual proposal exists, so this discussion remains speculative.

Decorators and contract assertions fundamentally differ and are essentially orthogonal. Decorators are an encapsulation mechanism for normal code: they run exactly once before and after a function and may perform arbitrary operations. Contract assertions, by contrast, are *ghost code*: they may run once, multiple times, or not at all ([P3264R1]) and are redundant to the correctness of the program. Further, precondition and postcondition assertions exist precisely at the caller-callee boundary. Any decorator facility would need to define with equal clarity how it relates to that boundary.

Concern 16: P2900 should be composed from other features

Summary

A recurring theme in [P3829R0] is the suggestion that contract assertions could somehow be composed from a set of more fundamental features that can be standardised individually.

Discussion Status

Similar decompositions were proposed to SG21 by [P1893R0] and/or suggested as ideas in EWG but achieved no consensus as the basis for a proposal that meets the use cases P2900 was pursuing. Relaxation of the ODR was polled by EWG in Hagenberg, with consensus against. The specific decomposition proposed in [P3829R0] has not been explicitly discussed in WG21; otherwise, no new information has been presented since Hagenberg.

Response

The decomposition proposed in [P3829R0] is an incomplete proposal that has not seen real discussion or specification nor been demonstrated to satisfy the requirements P2900 has been built to satisfy. Some of the features [P3829R0] proposes as part of its decomposition, such as decorators, are actually orthogonal to assertions; others, such as a relaxation of the ODR, have been explicitly rejected by EWG.

Details

According to [P2000R4], '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'. P2900 clearly qualifies as 'in flight', whereas the decomposition sketch in [P3829R0] lacks the required detailed discussion.

The idea to redesign contract assertions as a composition of more primitive features was first proposed in [P1893R0] and subsequently shown to be inadequate for the real-world use cases for contract assertions ([P1995R1]). [P3829R0] similarly mischaracterises both the nature and the intended use of contract assertions. Its proposed decomposition consists of 'safe ODR exclusions', function decorators, and lazy execution. None of these components, individually or in combination, capture the semantics that contract assertions require.

The first argument made by [P3829R0] is that contract assertions depend upon some form of ODR relaxation that might be broadly useful. The claim being made is that a function has a different definition if it is compiled with different known semantics for the same contract assertions. However, this claim is hard to support when no textual difference or no difference in intention exists between the function definitions being processed within each translation unit.

In the model of P2900, compilers are given leeway to optimise code based on the chosen semantic as soon as the compiler knows that final choice. To remove any runtime overhead for ignored contract assertions in the majority of common builds (that perform no link-time optimisations), we must allow that choice to be set during the compilation step, well before link time. For a more complete description of this procedure and the optimisation reasoning, see Section 2.2 of [P3321R0]. The belief that contract assertions require relaxing the ODR seems to be based on incorrect assumptions made by GCC about linking behaviour and derefinement (see Concern 4), because the optimisation that code undergoes once the semantic is known is a refinement of the potential behaviours it could have if a different semantic had been chosen.

Further, contract assertions, together with erroneous behaviour ([P2795R5]), are distinguished from all other C++ features in that they provide a principled way to reason about *incorrect* yet well-

defined programs. This new categorisation within the language requires the compiler to be free to inject or omit correctness-checking code — a very contract-specific and non-decomposable property. For the same reason, contract assertions are orthogonal to function decorators (see Concern 13 and Concern 14), despite a superficial resemblance, because precondition and postcondition assertions can lead to code being executed before and/or after a function call.

The idea to use lazy execution of function arguments or expressions as a building block for other language features came up several times in EWG (e.g., during discussion of [P1774R8] and in the discussion of [P2826R2], which might evolve to support expression aliasing in its next iteration). Yet no proposal for such a feature has matured to a state ready for adoption consideration. If such a proposal were made and standardised independently, `contract_assert` could be respecified backward-compatibly to accommodate such a mechanism. The possibility of adding lazy execution later does not undermine the design of P2900.

Finally, the proposed decomposition seems to obstruct having a global contract-violation handler that can be shared across libraries. [P3829R0] hastily dismisses the idea of a global handler, but as discussed above (see Concern 9), a global contract-violation handler is essential for using contract assertions at scale and is a primary motivation for standardising them.

Concern 17: P2900 has insufficient deployment experience

Summary

[P3849R0], [P3851R0], [ES-049], [US 26-051], [FR-004-053], [FR-005-054], and [FI-071] raise the concern that contract assertions as specified in P2900 lack sufficient implementation, in-field deployment, and usage experience.

Discussion Status

This concern repeats earlier objections ([P3173R0], [P3506R0], [P3573R0]) already considered repeatedly in EWG. No new information has been presented since.

Response

Expecting a reasonable level of implementation experience before standardising a novel language feature is good engineering practice that we strongly support. P2900 has been fully implemented in two major compilers. While it has not been deployed to production, neither has any other major language feature adopted by C++ in any previous or current Standard. The various component pieces and fundamental semantics from which P2900 is built, however, have decades of experience in a wide variety of forms, showing that they are both needed and sound.

Details

Two complete implementations of P2900 exist in GCC and Clang ([P3460R0]) and are publicly available (including on Compiler Explorer), with upstreaming in progress. With these implementations nearly complete, contract assertions are much closer to being merged to mainline branches

of these compilers than are other significant C++ features (such as modules) at this point in the standardisation process. The implementers reported no significant implementation challenges and described P2900's specification as 'clear and implementable'.

Deployment experience with the *entire* feature set of P2900 is admittedly still limited, in particular with `pre` and `post`. Realistically, significant real-world experience with the complete feature cannot be gained until it is part of the ISO C++ Standard. A TS or whitepaper (as suggested previously in [P3265R3] and now again in [P3849R0], [P3851R0], [ES-049], [FI-071]) would not lead to widespread use; past experience with concepts and modules shows that *language* TSes do not drive adoption of novel features ([P3269R0], [P3276R0], [P3297R0], [P4000R0]), only standardisation and shipping implementations in major compilers do. Requiring significant deployment experience in advance of standardisation creates a chicken-and-egg problem: applied consistently, it would have prevented most language features of modern C++ — many far more complex than contract assertions (see Concern 13) — from ever being realised.

At the same time, we *do* have substantial deployment experience with *components* of P2900. In the LLVM codebase, experiments have been performed to implement libc++'s internal `_LIBCPP_ASSERT` macro with `contract_assert` — including `const`-ification (see Concern 8), which uncovered multiple `const`-correctness bugs ([P3460R0]). The GCC implementation of P2900 has been extensively exercised in the BDE codebase ([P3336R0]), as has the Clang implementation since the publication of that report. As mentioned in Concern 3, Boost.Build has already cleanly and quickly added support for P2900 on top of the available GCC and Clang implementations.

Most importantly, libc++'s hardening modes provide extensive real-world experience with three of the four contract-evaluation semantics in P2900 (*ignore*, *enforce*, and *quick-enforce*), including selection via compiler flags and support for mixed mode. The implementers of libc++ have recently added the option of selecting the *observe* semantic as a necessary feature to roll out to some large codebases. These facilities have shipped to millions of users. Finally, the library API for contract-violation handling in P2900 closely mirrors equivalent facilities already used in the internal codebases of multiple large companies.

Some dismiss this experience as irrelevant because the shipped version of libc++ uses assertion macros rather than `pre` and `post`. However, the flexible evaluation-semantics model of P2900 and its realisation in compilers and linkers is entirely orthogonal to syntax. The design of P2900 and the deployment of libc++ hardening modes have informed each other directly: *quick-enforce* was added to P2900 as a result of experience in libc++ ([P3191R0]), and standard-library hardening is now defined in terms of P2900, both in the published libc++ documentation[4] and in the specification for standard-library hardening ([P3471R4]) slated for release in C++26.

[4] Available at `https://libcxx.llvm.org/Hardening.html`.

Concern 18: Standard-library hardening should not depend on Contracts

Summary

[FR-001-014], [US 3-015], [US 61-112], and [FR-010-113] ask for standard-library hardening to be specified independently from contract assertions due to a lack of deployment experience with the latter. [P3878R0] raises concerns that contract assertions are not a good fit for standard-library hardening because they are not sufficient to implement it. In addition, it asks to disallow the *observe* semantic on hardened preconditions.

Discussion Status

Standard-library hardening was rebased on top of P2900 in revision [P3471R1]; SG23 had very strong consensus for this approach. In the same revision, *observe* was added as an option. The concerns in [P3878R0] that a hardened standard library may have to target multiple compilers is new information; its concern that P2900 on its own is not insufficient to implement standard-library hardening is not.

Response

Both the libc++ and libstdc++ implementation currently being planned once contracts are available are conforming implementations of C++26 standard-library hardening on top of P2900. An attempt to modify this specification will not change this fact, but merely add more implementation-defined behaviour to hardened preconditions that does not make for a coherent language. Allowing the *observe* semantic is an important requirement in order to deploy these implementations at scale; however, since this allows the implementation to continue execution past a checked standard-library precondition, it seems sound to use a different term than "hardening" to describe these semantics.

Details

The primary concern that has been expressed regards whether we believe that standard-library hardening can and should be integrated with the contract-violation handling mechanism. While it is true that the specification of standard library hardening, as it stands now, cannot be implemented *purely* in terms of the basic feature set in C++26 Contracts, it more than sufficiently supports multiple implementation strategies that work seemlessly *on top* of C++26 Contracts.

To implement hardening in the standard library using `pre`, `post`, or `contract_assert`, a mechanism for grouping contract assertions — labels ([P3400R1]) or some equivalent feature — and configuring such groups independently is required. This is because a contract assertion used in a hardened standard library should not be subject to the same options controlling evaluation semantic as other user-defined contract assertions, but instead be based on whether a hardened standard library has been selected.

Standard Library implementations are free to depend on vendor extensions for this functionality until [P3400R1] is adopted, or to base their behaviors on preprocessor control and emulate the *quick-enforce* semantic. In particular, we expect libc++ (based on experimental implementations)

use contract assertions with implementation-defined attributes to facilitate that control. We expect libstdc++ to terminate when a hardened precondition is violated, which is a valid implementation of a contract assertion with the quick-enforce semantic.

Implementations are free to recognize those contract assertions that are present for hardening in any way they choose, and thus base the semantics of those assertions on a compiler flag such as `-fhardened` instead of other configuration options that might control other user-defined contract assertions. This approach can be done entirely in the compiler without any interaction with a standard library implementation other than agreeing on the use of `contract_assert` or `pre` for hardened preconditions.

A preprocessor-based solution that emulates *quick-enforce* can be used as a fallback by any Standard Library that is being used on a platform where it does not have the tools to provide better QoI.

Importantly, the specification of standard library hardening requires that a violation of a hardened precondition behaves *as if* a contract assertion had been violated, but this does not require the use of a literal contract assertion (the syntactic construct) for that purpose.

Without a replacement specification, suggestions to change to an otherwise-undefined term *runtime violation* are simply suggestions to increase the amount of implementation-defined behavior in the standard drastically, see Concern 6. Allowing unrelated methods of handling the notification that there is a bug in a program is actively user-hostile and something that P2900's design has been focusing on avoiding.

RU-016 expressly requests that standard-library hardening remain based on C++26 contract assertions. One of the requirements behind this request is that the *enforce* semantic can be used instead of the *quick-enforce* semantic so that logging and shutdown implemented in a centralised contract-violation handler can be employed for standard-library preconditions.

In [P3878R0] two major salient points can be extracted. First, standard-library hardening should not allow continued execution after a violation. Though this was not what was adopted originally with the standard-library hardening proposal ([P3471R4]) such a clarification on the intention of hardening and the reduction to allow only the *enforce* and *quick-enforce* semantics is a sound decision that the committee could make. Implementations could still freely allow the needed use of the *observe* semantic for such preconditions as long as they don't do so while simultaneoulsy claiming that the library is hardened.

The second claim in [P3878R0] is that implementations will find it challenging to use compiler-specific extensions to support standard-library hardening. Though this might be true, Standard Library implementations already communicate any number of subtleties with their underlying implementations using compiler builtins or implementation-specific macros, and this kind of interaction is not new, novel, or particularly challenging.

3 Conclusion

In this paper, we have systematically examined each concern raised in recent papers and NB comments regarding the current specification of contract assertions in the C++26 working draft. For each concern, we have provided:

- a clear and technically accurate summary of the concern
- the history of its consideration during development of P2900 in SG21 and EWG
- a description of how each concern is addressed in C++26
- pointers to future (in-progress) proposals that will expand the use cases covered and further address any remaining concerns

We hope that the detailed analysis provided herein demonstrates that the latest objections are neither new nor indications of fundamental flaws in the design of P2900. This design has already achieved strong consensus within WG21 and deserves to remain in C++26 as one of its cornerstone features.

Acknowledgments

We thank Harald Achitz, Brian Bi, Hartmut Kaiser, Andrzej Krzemieński, Jens Maurer, Herb Sutter, and Anton Polukhin for reviewing a draft of this paper and providing valuable feedback.

We thank Lori Hughes for volunteering to help make this paper as direct and to the point as possible. Any remaining mistakes in this paper are the responisbility of the primary authors.

Appendix: NB Comments overview

The table below lists all NB comments being addressed in this paper. The full wording of each NB comment and its proposed change can be found in [N5028].

NB number	Brief description	Response
[SE]	There are concerns over P2900 as described in [P3849R0].	Concerns 3, 7, 17
[ES-046]	Critical checks may be elided, opening up a security vulnerability that enables new supply-chain attacks.	Concern 4
[ES-047]	Observing multiple dependent assertions may lead to undefined behaviour.	Concern 10
[ES-048]	P2900 does not enforce consistent contract-evaluation semantics across translation units.	Concern 2
[ES-049]	P2900 lacks deployment experience, user experience, and implementation experience.	Concern 17
	P2900 lacks build system experience.	Concern 3
[ES-050]	P2900 exhibits 'several serious problems' as described in [P3851R0].	Concerns 2, 4, 8, 10, 14, 17
[US 26-051]	P2900 has too much implementation-defined behaviour.	Concern 6
	P2900 lacks deployment experience.	Concern 17
[US 25-052]	P2900 is 'not ready for standardization' as described in [P3829R0] and [P3835R0].	Concerns 1, 2, 4, 5, 6, 9, 13, 14, 15, 16, 17
[FR-004-053]	P2900 is insufficiently tested and lacks user experience.	Concern 17
[FR-005-054]	P2900 has too much implementation-defined behaviour.	Concern 6
	P2900 lacks deployment experience.	Concern 17
[ES-055]	`const`-ification is a concern for teachability, maintainability, and simplicity.	Concern 8
[RO 2-056]	P2900 relies excessively on implementation-defined behaviour.	Concern 6
	P2900 'does not allow a function to be made safe' because the evaluation semantics can be altered outside of the code.	Concern 1
[CZ 4-058]	`const`-ification makes automatic assertion insertion by tooling harder.	Concern 8

[FI-071]	P2900 lacks implementation and deployment experience.	Concern 17
	P2900 'is not a safety facility'.	Concern 1
	Treating exceptions thrown from contract-predicate evaluation as contract violations is not feasible on the Microsoft ABI.	Concern 11
	P2900 does not provide 'serious support' for static analysis tools.	Concern 12
[ES-072]	P2900 lacks support for `pre` and `post` on virtual functions.	Concern 14
[ES-073]	A non-reference parameter of a function in a post-condition assertion 'magically becomes `const`'.	Concern 8
[ES-074]	P2900 lacks support for `pre` and `post` on function pointers.	Concern 14
[FR-001-014] [US 3-015] [US 61-112] [FR-010-113]	Standard-library hardening should be specified independently from contract assertions	Concern 18

Bibliography

[CodeQL] "SMT constraint solving in CodeQL with Z3"
https://github.com/advanced-security/codeql-contracts-smt-z3

[Das16] Sanjoy Das, "Inter-Procedural Optimization and Derefinement"
https://www.playingwithpointers.com/blog/ipo-and-derefinement.html

[Doumler24] Timur Doumler, "Contracts for C++ explained in 5 minutes"
https://timur.audio/contracts_explained_in_5_mins

[Lippincott24] Lisa Lippincott, "Perspectives on Contracts for C++". CppCon, September 2024 YouTube

[Martin25] Peter Martin and Mike Fairhurst, "Catching Bugs Early: Validating C++ Contracts with Static Analysis". CppCon, September 2025 YouTube

[Meyer97] Bertrand Meyer, *Object-Oriented Software Construction* (Prentice Hall, 1997)

[Nash25] Phil Nash, "Contracts and Safety for C++26: An expert roundtable". C++ London Meetup YouTube

[Sutter04] Herb Sutter and Andrei Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* (Boston, MA: Addison-Wesley Professional, 2004)

[Sutter19] Herb Sutter, "De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable". CppCon, September 2019 YouTube

[Sutter24] Herb Sutter, "C++ safety, in context". Sutter's Mill
https://herbsutter.com/2024/03/11/safety-in-context/

[Sutter25] Herb Sutter, "The Joy of C++26 Contracts (and Some Myth-Conceptions)". CppCon, September 2025 YouTube

[N1613] Thorsten Ottosen, "Proposal to add Design by Contract to C++", 2004
http://wg21.link/N1613

[N3051] Doug Gregor, "Deprecating Exception Specifications", 2010
http://wg21.link/N3051

[N4378] John Lakos, Nathan Myers, Alexei Zakharov, and Alexander Beels, "Language Support for Contract Assertions", 2015
http://wg21.link/N4378

[N5007] Nina Ranns, "WG21 02/2025 Hagenberg Minutes of Meeting", 2025
http://wg21.link/N5007

[N5028] Herb Sutter, "C++26 CD summary of voting and comments", 2025
http://wg21.link/N5028

[P0003R5] Alisdair Meredith, "Removing Deprecated Exception Specifications from C++17", 2016
http://wg21.link/P0003R5

[P0542R5] J. Daniel Garcia, “Support for contract based programming in C++”, 2018
http://wg21.link/P0542R5

[P1494R5] S. Davis Herring, “Partial program correctness”, 2025
http://wg21.link/P1494R5

[P1774R8] Timur Doumler, “Portable assumptions”, 2022
http://wg21.link/P1774R8

[P1823R0] Nicolai Josuttis, Ville Voutilainen, Roger Orr, Daveed Vandevoorde, John Spicer, and Christopher Di Bella, “Remove Contracts from C++20”, 2019
http://wg21.link/P1823R0

[P1893R0] Andrew Tomazos, “Proposal of Contract Primitives”, 2019
http://wg21.link/P1893R0

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, “Contracts — Use Cases”, 2020
http://wg21.link/P1995R1

[P2000R4] Roger Orr, Howard Hinnant, Roger Orr, Bjarne Stroustrup, Daveed Vandevoorde, and Michael Wong, “Direction for ISO C++”, 2022
http://wg21.link/P2000R4

[P2300R10] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, and Bryce Adelstein Lelbach, “`std::execution`”, 2024
http://wg21.link/P2300R10

[P2680R1] Gabriel Dos Reis, “Contracts for C++: Prioritizing Safety”, 2023
http://wg21.link/P2680R1

[P2695R1] Timur Doumler and John Spicer, “A proposed plan for contracts in C++”, 2023
http://wg21.link/P2695R1

[P2712R0] Joshua Berne, “Classification of Contract-Checking Predicates”, 2022
http://wg21.link/P2712R0

[P2751R0] Joshua Berne, “Evaluation of *Checked* Contract-Checking Annotations”, 2023
http://wg21.link/P2751R0

[P2751R1] Joshua Berne, “Evaluation of *Checked* Contract-Checking Annotations”, 2023
http://wg21.link/P2751R1

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, “A Bold Plan for a Complete Contracts Facility”, 2024
http://wg21.link/P2755R1

[P2780R0] Ville Voutilainen, “Caller-side precondition checking, and Eval_and_throw”, 2023
http://wg21.link/P2780R0

[P2795R5] Thomas Köppe, “Erroneous behaviour for uninitialized reads”, 2024
http://wg21.link/P2795R5

[P2811R7] Joshua Berne, “Contract-Violation Handlers”, 2023
http://wg21.link/P2811R7

[P2826R2] Gašper Ažman, “Replacement functions”, 2024
http://wg21.link/P2826R2

[P2877R0] Joshua Berne and Tom Honermann, “Contract Build Modes, Semantics, and Implementation Strategies”, 2023
http://wg21.link/P2877R0

[P2885R3] Timur Doumler, Joshua Berne, Gašper Ažman, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann, “Requirements for a Contracts syntax”, 2023
http://wg21.link/P2885R3

[P2899R1] Timur Doumler, Joshua Berne, Andrzej Krzemieński, and Rostislav Khlebnikov, “Contracts for C++ - Rationale”, 2025
http://wg21.link/P2899R1

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R6

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2025
http://wg21.link/P2900R14

[P2957R2] Andrzej Krzemieński, Iain Sandoe, Joshua Berne, and Timur Doumler, “Contracts and coroutines”, 2024
http://wg21.link/P2957R2

[P2996R13] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, and Dan Katz, “Reflection for C++26”, 2025
http://wg21.link/P2996R13

[P3071R1] Jens Maurer, “Protection against modifications in contracts”, 2023
http://wg21.link/P3071R1

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Contracts for C++: Support for virtual functions”, 2024
http://wg21.link/P3097R0

[P3098R1] Timur Doumler, Gašper Ažman, and Joshua Berne, “Contracts for C++: Post-condition captures”, 2024
http://wg21.link/P3098R1

[P3099R0] Timur Doumler, Peter Bindels, and Joshua Berne, “Contracts for C++: User-defined diagnostic messages”, 2025
http://wg21.link/P3099R0

[P3119R1] Joshua Berne, "Tokyo Technical Fixes to Contracts", 2024
http://wg21.link/P3119R1

[P3173R0] Gabriel Dos Reis, "[P2900R6] May Be Minimal, but It Is Not Viable", 2024
http://wg21.link/P3173R0

[P3191R0] Louis Dionne, Yeoul Na, and Konstantin Varlamov, "Feedback on the scalability of contract violation handlers in P2900", 2024
http://wg21.link/P3191R0

[P3204R0] Joshua Berne, "Why Contracts?", 2024
http://wg21.link/P3204R0

[P3261R2] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
http://wg21.link/P3261R2

[P3264R1] Ville Voutilainen, "Double-evaluation of preconditions", 2024
http://wg21.link/P3264R1

[P3265R3] Ville Voutilainen, "Ship Contracts in a TS", 2024
http://wg21.link/P3265R3

[P3267R1] Peter Bindels and Tom Honermann, "Approaches to C++ Contracts", 2024
http://wg21.link/P3267R1

[P3268R0] Peter Bindels, "C++ Contracts Constification Challenges Concerning Current Code", 2024
http://wg21.link/P3268R0

[P3269R0] Timur Doumler and John Spicer, "Do Not Ship Contracts as a TS", 2024
http://wg21.link/P3269R0

[P3271R1] Lisa Lippincott, "Function Types with Usage (Contracts for Function Pointers)", 2024
http://wg21.link/P3271R1

[P3276R0] Joshua Berne, Steve Downey, Jake Fevold, Mungo Gill, Rostislav Khlebnikov, John Lakos, and Alisdair Meredith, "P2900 Is Superior to a Contracts TS", 2024
http://wg21.link/P3276R0

[P3285R0] Gabriel Dos Reis, "Contracts: Protecting The Protector", 2024
http://wg21.link/P3285R0

[P3297R0] Ryan McDougall, Jean-Francois Campeau, Christian Eltzschig, Mathias Kraus, and Pez Zarifian, "C++26 Needs Contract Checking", 2024
http://wg21.link/P3297R0

[P3321R0] Joshua Berne, "Contracts Interaction With Tooling", 2024
http://wg21.link/P3321R0

[P3327R0] Timur Doumler, "Contract assertions on function pointers", 2024
http://wg21.link/P3327R0

[P3336R0] Joshua Berne, "Usage Experience for Contracts with BDE", 2024
http://wg21.link/P3336R0

[P3362R0] Ville Voutilainen, "Static analysis and 'safety' of Contracts, P2900 vs. P2680/P3285", 2024
http://wg21.link/P3362R0

[P3376R0] Andrzej Krzemieński, "Contract assertions versus static analysis and 'safety'", 2024
http://wg21.link/P3376R0

[P3386R1] Joshua Berne, "Static Analysis of Contracts with P2900", 2024
http://wg21.link/P3386R1

[P3400R1] Joshua Berne, "Specifying Contract Assertion Properties with Labels", 2025
http://wg21.link/P3400R1

[P3460R0] Eric Fiselier, Nina Dinka Ranns, and Iain Sandoe, "Contracts Implementors Report", 2024
http://wg21.link/P3460R0

[P3471R1] Konstantin Varlamov and Louis Dionne, "Standard Library Hardening", 2024
http://wg21.link/P3471R1

[P3471R4] Konstantin Varlamov and Louis Dionne, "Standard Library Hardening", 2025
http://wg21.link/P3471R4

[P3478R0] John Spicer, "Constification should not be part of the MVP", 2024
http://wg21.link/P3478R0

[P3499R1] Timur Doumler, Lisa Lippincott, and Joshua Berne, "Exploring strict contract predicates", 2025
http://wg21.link/P3499R1

[P3500R1] Timur Doumler, Gašper Ažman, Joshua Berne, and Ryan McDougall, "Are Contracts "safe"?", 2025
http://wg21.link/P3500R1

[P3506R0] Gabriel Dos Reis, "P2900 Is Still not Ready for C++26", 2025
http://wg21.link/P3506R0

[P3573R0] Bjarne Stroustrup, Michael Hava, J. Daniel Garcia Sanchez, Ran Regev, Gabriel Dos Reis, John Spicer, J.C. van Winkel, David Vandevoorde, and Ville Voutilainen, "Contract concerns", 2025
http://wg21.link/P3573R0

[P3578R1] Ryan McDougall, "What is "Safety"?", 2025
http://wg21.link/P3578R1

[P3582R0] Andrzej Krzemieński, "Observed a contract violation? Skip subsequent assertions!", 2025
http://wg21.link/P3582R0

[P3583R0] Jonas Persson, “Contracts, Types & Functions”, 2025
http://wg21.link/P3583R0

[P3591R0] Joshua Berne and Timur Doumler, “Contextualizing Contracts Concerns”, 2025
http://wg21.link/P3591R0

[P3829R0] David Chisnall, John Spicer, Gabriel Dos Reis, Ville Voutilainen, and Jose Daniel Garcia Sanchez, “Contracts do not belong in the language”, 2025
http://wg21.link/P3829R0

[P3835R0] John Spicer, Ville Voutilainen, and Jose Daniel Garcia Sanchez, “Contracts make C++ less safe – full stop”, 2025
http://wg21.link/P3835R0

[P3849R0] Harald Achitz, “SIS/TK611 considerations on Contract Assertions”, 2025
http://wg21.link/P3849R0

[P3851R0] J. Daniel Garcia, Jose Gomez, Raul Huertas, Javier Lopez-Gomez, Jesus Martinez, Francisco Palomo, and Victor Sanchez, “Position on contracts assertion for C++26”, 2025
http://wg21.link/P3851R0

[P3853R0] Ville Voutilainen, “A thesis+antithesis=synthesis rumination on Contracts”, 2025
http://wg21.link/P3853R0

[P3878R0] Ville Voutilainen, Jonathan Wakely, and John Spicer, “C++26 Contracts are not a good fit for standard library hardening”, 2025
http://wg21.link/P3878R0

[P3889R0] Harald Achitz, “A minimal solution for contracts, or, what is an MVP?”, 2025
http://wg21.link/P3889R0

[P3893R0] Mike Fairhurst, “The CppCon 2025 Talk on Contracts and CodeQL in Context”, 2025
http://wg21.link/P3893R0

[P3896R0] Andrzej Krzemieński, “Design goals for a contract support facility”, 2025
http://wg21.link/P3896R0

[P3909R0] Ville Voutilainen, “Contracts should go into a White Paper - even at this late point”, 2025
http://wg21.link/P3909R0

[P3910R0] Bengt Gustafsson, “Improving safety of C++26 contracts”, 2025
http://wg21.link/P3910R0

[P4000R0] Michael Wong, H. Hinnant, R. Orr, B. Stroustrup, and D. Vandevoorde, “To TS or not to TS: that is the question”, 2024
http://wg21.link/P4000R0

[SD4] “Practices and Procedures: The “How We Work” Cheat Sheet”
https://wg21.link/sd4