Prevent Undefined Behavior By Default

Document #:	P3558R1
Date:	2025-02-02
Project:	Programming Language C++
Audience:	SG23 (Safety), SG15 (Tooling), EWG (Evolution)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	John Lakos <jlakos@bloomberg.net>

Abstract

Managing the impact of undefined behavior on the functional safety and correctness of programs written in C++ is the key challenge of our time and one that WG21 must urgently address. In [P3100R1], a mechanism for clearly specifying most undefined behavior as the consequence of violated core-language preconditions, using the mechanisms of Contracts introduced in [P2900R13], provides a framework for improving the world's ability to write functionally safe and correct code in C++ with no need for compromises. We propose not only pursuing this route for *all* runtime-checkable undefined behavior, but also encouraging vendors to make checking these conditions the default in all implementations.

Contents

Revision History

Revision 1

- Elaborated on more concerns
- Added an alternate proposal to begin adoption with [P3599R0]

Revision 0

- Original version of the paper

1 Introduction

For many years, the C++ Standards Committee (WG21) — particularly the Contracts study group of that Committee (SG21) — has been working on defining a flexible and powerful framework that users can employ to diagnose and mitigate the impact of bugs in their software. The first major step forward produced by that effort is the Contracts MVP proposed in [P2900R13].

Undefined behavior, when encountered, is *always* a bug in a user's program — be it in the code they wrote or in a library they are using. As described in [P3100R1], most undefined behavior can be categorized as violations of *core-language preconditions* that are *implicitly* present wherever such undefined behavior occurs today. In addition, as of today and with no need for extra permission to do so, each of these preconditions is already being effectively evaluated with the *assume* semantic.

[P3100R1] proposes making this specification explicit and encouraging vendors to provide users with consistent mechanisms to select other semantics for these checks (e.g., *enforce*, *quick-enforce*, or *observe*), allowing those users to dramatically increase safety (of all sorts) albeit with a potential cost in runtime performance.

We propose taking this approach a step further, encouraging vendors to make enforcing these preconditions the default everywhere, as is done for library-level contract assertions with [P2900R13], to reap several associated game-changing benefits.

- The C++ ecosystem will, out of the box, be safer to use on any platform.

  Keeping active, by default, any passive safety system that is not mandated in absolutely every case, such as air bags, is important unless a compelling local reason exists to disable it. For implicit contract assertions, this need is especially acute for less-experienced developers, particularly students learning C++, who will have absolutely no idea that they should first flip some esoteric compiler switch to enable this immensely valuable implicit core-language precondition enforcement. Note that this automatic language safety is a property Rust users rely on but, unlike with C++26, cannot selectively disable to reclaim performance.

  With each runtime-checkable mistake they make, novices will be informed, via meaningful diagnostics, immediately and automatically of their error, rather than relying on first digesting enough documentation to configure their compiler appropriately. Such immediate feedback forces exposure of a wide variety of defects very early in a user's learning career and in the lifecycle of the software they write, which in turn facilitates rapid learning by the person who created the bug without the need to engage additional tooling, and avoids the tedium and frustration that can lead C++ novices to migration to other languages.

- Anyone maintaining a C++ program that is significantly impacted by any performance costs resulting from these additional runtime checks will already be more than capable of choosing compiler configurations that fine-tune performance for their needs and can do so just as easily with any checks that are not needed. Importantly, the non-performance-sensitive uses of that same source code, in unit tests and development environments, will still easily benefit from checking these assertions and reduce the bugs and security vulnerabilities in the eventually deployed high-performance release builds.

- The decision to opt *out* versus *in* to these protections is not at all symmetric. If we don't know we have a bug, nothing will tell us to turn on checking. If checking is on by default, however, we will catch the bug regardless of whether we thought we needed checking. Similarly, if checking is opt in and we don't avail ourselves of it, we won't know if the impact is enough to make a difference. Conversely, if checking is on by default, all the code will be guarded unless or until clear and compelling evidence indicates that some code is too slow, at which point that theory can be empirically measured to see if indeed the checking is causing the slowdown. In other words, when in doubt or when the cost is not meaningful, we want the code to start out protected (by default) so that as much of the code as possible continues to remain protected until empirical evidence shows that needed runtime permanence is compromised.

- Other than how the resulting violations are presented[1] to the contract-violation handler, nothing about the proposed core-language preconditions in [P3100R1] results in any changes to how C++ source code is written. All benefits of such checks apply not only to newly developed software written in C++26 or C++29, but also to all legacy C++ programs written to conform to any Standard just by recompiling with a new compiler that has chosen to enable those checks by default.

Most importantly, for those concerned with the runtime overhead of defining undefined core-language behavior, the approach proposed in [P3100R1] is optimal.

- The decision to check such preconditions is not made in source code, and code is not written differently because the behavior has been defined. Therefore, the same libraries that work for users who want safety will work exactly as well for users who need optimal performance.

- When possible, a well-defined fallback behavior can be specified, reducing risk significantly even when bugs are not found and fixed, realizing — with no need for changes to source code — the great benefits of *erroneous behavior*, as introduced in [P2795R0]. Surprisingly, this design can be specified without any need to define erroneous behavior in the language specification; implicit contract assertions are introduced, some previously undefined behavior is defined, and the benefits of erroneous behavior are kept.

Three questions remain.

[1]In [P3100R0], violations of a core-language precondition can (if the *observe* or *enforce* semantic is chosen) invoke the contract-violation handler, in which case the `kind` on the `contract_violation` object will be `implicit` and other values will be used to indicate the specific core-language precondition that was violated. Note that these additional enumerators represent the only explicit API change proposed in [P3100R1].

1. What concerns does the Committee have regarding this approach to handling undefined behavior?
2. What should be done with profiles that seek to introduce runtime checks for core-language preconditions?
3. What parts of these proposals are necessary to achieve a meaningful improvement in the safety of C++ in a meaningful timeframe?

2 Concerns

Any compiler change carries concerns.

- **Overhead of Checking** — The overhead of checking core-language preconditions is going to vary from one such precondition to the next. Many experiments in compilers have been conducted with varying results for each kind of undefined behavior for which we would consider introducing checks. Only real implementation experience will tell us how often each kind of check will need to be reverted to the current *assume* semantic, but in general, experience has shown that most such checks do not introduce sufficient overhead to impact the utility of the typical program.

- **Potentially Throwing Contract-Violation Handlers** — Much concern has been raised, such as in [P3541R0], about the impact that throwing violation handlers will have on the language. In particular, by allowing a violation of a core-language contract to emit an exception when the installed contract-violation handler throws (since the predicate of such a precondition will clearly never throw), we open the door to exceptions escaping from expressions from which they previously never would have escaped. This behavior can be surprising and can leave code in a bad state, yet the current behavior is undefined and leaves the program in an even worse state.

  Altering undefined behavior based on build modes is not only conforming, but a fact of life. Consider the case of defective code that nonetheless works as intended in unoptimized mode but fails specularity with even the `-O1` compiler flag. We simply cannot require that optimization not alter the observed behaviors of programs whose behavior is undefined. That said, the range of behaviors seen for many forms of undefined behavior is, in practice, reasonably bounded and certainly does not include throwing an exception.

  A suggestion has been made, however, that legacy code might be dependent on certain operations, such as null pointer dereferences, leading to core dumps today. While depending on such behaviors as if they were guaranteed is unwise, Hyrum's law somewhat forces us to seriously consider any change to that behavior. Note that the default behavior we recommend — *enforcing* the implicit contract assertions — provides exactly that behavior. Only expert users decide to install a throwing contract-violation handler that will encounter any new and surprising control paths, and those users most need such flexibility.

- **Overhead of Exception-Handling Scaffolding** — Potentially throwing violation handlers also require extra code to be generated to handle exceptions escaping. We strongly suggest that compilers implement switches for users to, as a global build-time option, require a nonthrowing violation handler and produce a translation unit that has none of that related overhead. Such a switch need not be used on all translation units that are being built; the compiler and linker, however, must enforce that any translation unit built this way results in requiring that the installed violation handler itself be `noexcept`. Of course, any platform that follows the recommended practice for how the default contract-violation handler behaves will be providing a contract-violation handler that simply cannot throw.

- **Undefined Behavior That Is Not Erroneous** — In some environments, behavior is selected for undefined behavior that is desired and designed for, and the users of such environments simply do not consider that behavior to be incorrect. Consider, for example, some users who might use the `-fwrapv` arguments to GCC, which defines the behavior of signed integer overflow as wrapping arithmetic ($\mathbb{Z}_{2^N}$, like unsigned integers). Some of those users might be doing so simply to mitigate the risks of overflow, and others might have the intent of wrapping and depending on that wrapping behavior.

  With each undefined behavior we seek to treat with the approach in [P3100R1], we must determine if we have consensus to always treat that behavior as erroneous or to instead leave it undefined. Other options to consider might be to make the determination of whether an implicit precondition is present implementation defined because that choice retains the full gamut of freedom that implementations and users have today.

- **Runtime Risks of Enforcing** — When new checks are introduced into a running program that previously did not include those checks, any violations will suddenly be diagnosed and result in termination if the chosen evaluation semantic is *enforce*. When a program had undefined behavior previously yet was still performing its duties correctly, this can result in a highly unpleasant experience with new language upgrades.

  This concern, however, is a large part of the reason for two of the evaluation semantics that are proposed as part of [P2900R13]: *ignore* and, more importantly, *observe*. By *ignoring* implicit preconditions, the existing behavior can be completely maintained. By *observing* them, however, diagnostics can allow for identifying the bugs in a nonfatal fashion while the existing behavior is also largely maintained, resulting in not only stable production behavior, but a path toward identifying and fixing the bugs before an unrelated change does turn the benign into the disastrous.

3 Profiles

Profiles — particularly the core-safety profiles proposed in [P3081R1] — have been aiming to provide two distinct features combined in a single specification.

1. Introduce restrictions on what is considered well formed C++ (e.g., restrictions on allowed language constructs) in a particular translation unit or at a particular scope.
2. Introduce runtime checks of certain core-language preconditions.

The first feature is eminently useful and a potentially powerful tool for improving the quality of C++ software.

The second group of features proposed for Profiles, however, is actively harmful in three ways.

1. We actively want all runtime checking in C++ — implicit core-language and explicit source-code-level contract assertions — to be on by default.
2. Profiles would redundantly intrude on how implicit preconditions are proposed to be controlled in [P3100R1].
3. Once we put something in a profile, the behavior of that check would override external settings, thereby undermining the build system's ability to control and tailor the safety and performance trade-offs for C++ software components, subsystems, and programs — let alone generally reusable libraries — as they move through their lifecycle.

We, therefore, propose that Profiles focus entirely on locally restricting the language to better subsets, while all control over runtime checks be moved out of Profiles and each have the *enforce* semantic by default, with compilers providing the appropriate configuration options to opt out of such checks if and as needed.

Proposal 1: Leave Runtime Checking Out of Profiles

Do not have Profiles introduce contract assertions or mandate the semantic with which any contract assertions are evaluated.

4 Proposal

The question then is what should be done immediately.

- Standardizing the enumerators proposed by [P3100R1] is the only part of the proposal that has any user-facing API change that would be beneficial to include in C++26.
- Even without those changes, compilers can still begin to implement runtime checks of core-language preconditions because any violations so detected are fully *undefined behavior* today and because the design in [P2900R13] has been carefully and explicitly crafted to support vendor extensions that introduce new enumeration values and values from future Standards being back-ported.
- The first group to review [P3100R1] was a joint session of SG21 (Contracts) and SG23 (Safety) in Wrocław. After this initial discussion, unanimous consent was reached to pursue the direction of [P3100R1] for handling undefined behavior for *all* core-language operations.

  SG21, Wrocław, 2024-11-22, Poll 6

  We support the direction of P3100R1 and encourage the authors to come back with a fully specified proposal.

SF	F	N	A	SA
19	6	0	0	0

  Result: Consensus

With no specific ship vehicle required, we need only to continue actively pursuing implicit core-language precondition checking and to avoid any attempt to invent source-code syntax for that purpose in C++26. Once labels ([P3400R0]) become available, we can use that needed flexibility to provide *both* command-line and in-source control over both implicit and explicit contract assertions.

Proposal 2: Pursue [P3100R1] to Completion

Continue to pursue a more complete specification of [P3100R1], and adopt it after [P2900R13].

Alternatively, if a more aggressive move to deliver implicit contract support for C++26 is desired, we have prepared [P3599R0], a complete proposal that contains a minimal subset of [P3100R1] — namely, the same set of checks that are proposed to be part of Profiles in [P3081R1] — and that is ready to propose for adoption into the working draft.

Proposal 3: Adopt [P3599R0]

Begin the process of introducing implicit contract assertions by adopting [P3599R0] after [P2900R13].

In addition, what default we recommend for the semantic of implicit core-language preconditions is another consideration and is important for both the perception of the safety of the C++ language and the user experience of novices to the language.

A recommended default is, of course, not enforceable, but the world pays attention to the best practices we place in the Standard, so that is the vehicle to use to suggest an appropriate default.

Proposal 4: Recommend Enforcement

When adopted, [P3100R1] should recommend a default evaluation semantic, when nothing else is specified, of *enforce* for all core-language preconditions.

5 Conclusion

For various reasons that have been covered extensively elsewhere, C++ is at an inflection point with regard to safety and its own future. We believe that pursuing core-language precondition checks, on top of the Contracts MVP, can mitigate the perception of C++ as an unsuitable tool for large-scale system software and provide real value.

Importantly, the simple act of pursuing these designs, seeing them implemented, and deploying them at scale will realize immediate benefits, without the effort and risk associated with adding new syntax or library components to the language.

Acknowledgments

Thanks to Mungo Gill and Timur Doumler for reviewing this paper.

Thanks to Lori Hughes for providing editorial feedback.

Bibliography

[P2795R0] Thomas Köppe, “Correct and incorrect code, and "erroneous behaviour"”, 2023
http://wg21.link/P2795R0

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2025
http://wg21.link/P2900R13

[P3081R1] Herb Sutter, “Core safety profiles for C++26”, 2025
http://wg21.link/P3081R1

[P3100R0] Timur Doumler, Gašper Ažman, and Joshua Berne, “Undefined and erroneous behaviour are contract violations”, 2024
http://wg21.link/P3100R0

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne, “Undefined and erroneous behaviour are contract violations”, 2024
http://wg21.link/P3100R1

[P3400R0] Joshua Berne, “Specifying Contract Assertion Properties with Labels”, 2025
http://wg21.link/P3400R0

[P3541R0] Andrzej Krzemieński, “Violation handlers vs `noexcept`”, 2024
http://wg21.link/P3541R0

[P3599R0] Timur Doumler and Joshua Berne, “Initial Implicit Contract Assertions”, 2024
http://wg21.link/P3599R0