Design considerations for always-enforced contract assertions

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.com)
Gašper Ažman (gasper.azman@gmail.com)
Oliver Rosten (oliver.rosten@gmail.com)
Lisa Lippincott (lisa.e.lippincott@gmail.com)
Peter Bindels (dascandy@gmail.com)

Document #: P3912R0
Date: 2025-12-15
Project: Programming Language C++
Audience: EWG

Abstract

To address concerns raised by NB comment [RO 2-056], EWG has requested a proposal introducing a new syntax for contract assertions that are *always enforced*. This idea was presented as a small addition to C++26. It overlaps with the more generic *labels* feature [P3400R2], which is currently being pursued for C++29.

The authors of [RO 2-056] have requested assistance in identifying the technical concerns that must be considered when proposing to add this functionality. To that end, this paper lays out a necessary set of design questions that any proposal for always-enforced contract assertions would need to answer.

Contents

1 Introduction

In C++, *assertions* can be characterised by the property that their evaluation semantics are determined independently of the source code. This is true of the C `assert` macro (via `NDEBUG`), most user-defined assertion macros, and *contract assertions* as specified in the current C++26 working draft [N5014]. The initial contracts feature set adopted for C++26 provides four evaluation semantics — *ignore*, *observe*, *enforce*, and *quick-enforce* — but intentionally does *not* provide a way to configure the evaluation semantic in source code.

NB comment [RO 2-056] and its accompanying paper [P3911R0], however, consider it a key requirement that contract assertions can be specified in source code as *always enforced*. That is, they must provide a language-level guarantee that execution cannot continue past a failed check. The NB comment requests that such a feature be added to C++26 before it is finalised. The suggested approach is to introduce a new syntax — such as `pre!(x)` — to denote a contract assertion that can never be *ignored* or *observed*, in contrast to the existing syntax `pre(x)`, which may be evaluated with any of the four standard evaluation semantics.

An extension to contract assertions proposed in [P3400R2] for C++29 introduces a generic framework for controlling the evaluation semantics of contract assertions in source code via *assertion-control objects* (also referred to as *labels*). This framework enables a wide range of additional use cases for contract assertions and, as a result of one of those use cases, also introduces support for always-enforced assertions. Under that proposal, such assertions can be written as `pre<always_enforced>(x)`, where `always_enforced` is an appropriately named and specified assertion-control object — either user-defined or provided by the standard library. While [P3400R2] has undergone multiple rounds of review in the now-defunct SG21, it has not yet been considered by EWG and is therefore not ready for C++26.

At the November 2025 meeting in Kona, EWG decided to pursue a feature along the lines of `pre!(x)` for C++26 in response to NB comment [RO 2-056]. Such a feature could eventually become syntactic sugar for a small subset of the functionality provided by [P3400R2].

Rushing a new feature with new syntax into C++26 during the NB comment phase runs counter to the established WG21 model for standardisation as described in [SD-4]. By the same token, the foundational contracts feature set in the C++26 working draft should not be delayed due to the absence of this additional functionality.

If EWG nevertheless proceeds with adding new syntax for always-enforced assertions in C++26, it is important that the design space be carefully explored and that the tradeoffs inherent in each design decision be well understood. Although [P3400R2] has not yet been approved for inclusion in Standard C++, it currently represents the most thoroughly explored design basis for such functionality, having already received significant effort and review.

This paper therefore presents a necessary set of design questions that any proposal for always-enforced assertions would need to answer—and which do not appear to be addressed with clear rationale in [RO 2-056] or its accompanying paper [P3911R0]. We hope this material will help structure the forthcoming EWG discussion.

2 Syntax

A particularly significant aspect of the proposed always-enforced assertions feature is the syntax. Even if it eventually becomes shorthand for a specific label introduced via the more generic labels feature [P3400R2], that shorthand syntax would become a permanent part of Standard C++. It therefore warrants careful consideration.

At the November 2025 WG21 meeting in Kona, the idea of adding an exclamation mark to the existing keywords — `pre!` instead of `pre` — to denote an always-enforced contract assertion gained some traction. In this paper, we use this syntax as a strawman. As discussed in Section 2.1 below, however, this syntax raises several design concerns, and alternative design directions may be preferable.

Any final proposal should include clear rationale for why a particular syntax is the right choice. The syntax for C++26 contract assertions was developed over several years and is backed by a detailed analysis of requirements, alternatives, and trade-offs (see [P2885R3]). Any additions or modifications to this design should arguably be justified with comparable rigour.

2.1 Usage of symbols

Should the new syntax consist of suffixing the existing keywords with a symbol? If so, which symbol should be use?

The exclamation mark (`!`) is particularly problematic as it conventionally denotes negation, and `pre!(x)` closely resembles `pre(!x)`, which is likely to confuse users. In addition, `!` has the alternative token `not`, meaning that `pre!(x)` could also be spelled as `pre not(x)`, introducing further oddity. Finally, in Rust, `identifier!` denotes a declarative macro; C++ may wish to reserve this syntactic space for a similar feature in the future.

While other symbols could be considered, no concrete alternatives have been proposed so far. Regarding the usage of symbols in general, when the current contracts syntax was developed ([P2885R3], [P2961R2]), WG14 was consulted and expressed a strong preference for a syntax that is structurally identical to a function call or function-like macro, in order to preserve the possibility of C compatibility. A symbol-based syntax would likely preclude this. It would also make it more difficult to conditionally disable the feature via the preprocessor on compilers that do not yet support it.

2.2 The choice of default

Should the default syntax — `pre(x)`, `post(x)`, and `contract_assert(x)` — denote regular or always-enforced assertions? Or should there be no default syntax at all?

In C++, as in most programming languages, the term "assert" is typically associated with checks that can be ignored and whose evaluation semantics are configured out-of-source, such as those in the C++26 working draft. Always-enforced checks are usually spelled differently. There are exceptions, most notably Rust, where `assert!(x)` denotes an

always-enforced check, while the more verbose syntax `debug_assert!(x)` denotes a build-time configurable (and thus ignorable) check. Note that *both* the always-enforced and the ignorable version end in an exclamation mark by virtue of being declarative macros.

During discussion of `pre!(x)`, it has been suggested that C++26 should instead use the default syntax `pre(x)` for always-enforced checks, and introduce a different syntax for ignorable checks. It has also been suggested that, if we are not sure about the correct default, the existing C++26 syntax should be made ill-formed and replaced with more explicit spellings for both flavours.

However, both approaches would constitute a substantial redesign of the existing C++26 contracts facility. Given that [P2900R14] was merged into the C++26 working draft with strong consensus and that the process has now reached the NB comment phase, such changes would run counter to established WG21 practice.

2.3 Relationship between regular and always-enforced syntax

How should the syntax for always-enforced assertions relate to that for regular assertions? Should it be shorter, longer, or approximately the same length? Should the two be structurally similar?

Regardless of which flavour of assertion — if any — should be the default, a proposal introducing always-enforced assertions must clarify how the syntax for always-enforced checks and that for ignorable checks should relate to each other.

Should the close relationship between the two be made explicit by using a slight variation of the same keyword (for example, `pre` versus `pre!`)? Or should always-enforced checks use a distinct keyword (for example, `pre` versus `enforce`) to emphasise that they represent a different feature?

Similarly, should always-enforced checks be shorter and easier to write than ignorable ones (for example, `pre_debug` versus `pre`), longer and more explicit (for example, `pre` versus `pre_always`), or roughly symmetric in structure (for example, `pre_debug` and `pre_always`)?

2.4 Forward-compatibility with labels

Should the syntax for always-enforced assertions be forward-compatible with labels?

If C++26 adopts a special-case syntax for always-enforced contract assertions and C++29 later adopts the full labels framework proposed in [P3400R2], the language may end up with two distinct ways to spell the exact same thing. While introducing shorthand for common operations is not inherently problematic, it does add permanent complexity with unclear benefit.

One way to avoid this situation would be to make the syntax fully forward-compatible with labels by committing now to the `pre<label>(x)` syntax of [P3400R2] and reserving an identifier such as `always_enforced` for a future standard label. The syntax

`pre<always_enforced>(x)` could then be introduced directly for C++26, avoiding the need for a special-case form like `pre!(x)`. However, such a syntax choice would also need to address questions such as whether use of the syntax depends on having included header `<contracts>` (as is the case for [P3400R2] labels).

Alternatively, `pre!(x)` could be given semantics intentionally distinct from a regular `pre` with a label that constrains the semantic to *enforce* and/or *quick-enforce*. The tradeoffs of such a direction are discussed in Sections 3.5 and 3.6. In that case, a clearly distinct syntax would arguably be more appropriate.

Either way, introducing any form of always-enforced assertion now would necessarily encroach on the design space of [P3400R2], complicating the future design of both features and potentially constraining their design choices in undesirable ways. Making an informed decision on these choices would require EWG review of [P3400R2], which does not appear feasible before C++26 ships. It therefore seems unwise to introduce a partially-overlapping feature on such an accelerated timeline.

3 Semantics

3.1 Evaluation semantics

Which evaluation semantics should be allowed for `pre!(x)` — *enforce*, *quick-enforce*, or both?

Both *enforce* and *quick-enforce* satisfy the core requirement articulated in [RO 2-056]: that execution must never continue past a failed check. Each evaluation semantic has legitimate use cases: *enforce* enables diagnostics and controlled shutdown via the contract-violation handler, while *quick-enforce* minimises the attack surface by avoiding any further code execution after a failed check.

If, however, using a single syntax `pre!(x)` permits both of these evaluation semantics, the choice between them becomes *implementation-defined* rather than specified in source code, and would typically be controlled via a build flag. Such a flag would be distinct from the build flags that control the evaluation semantic of regular assertions as it offers less flexibility by design. It is important to clarify whether such build-time configurability of `pre!(x)` aligns with user expectations, particularly given that some of the motivation for `pre!` appears to stem from discomfort with build-time configurations of assertions altogether.

3.2 Contract assertion kinds

To which contract assertion kinds should the new syntax apply — all of `pre`, `post`, and `contract_assert`, or only a subset?

This question matters because several of the design issues discussed below — such as mixing regular and always-enforced assertions, indirect calls, and caller-side versus

callee-side checking — only arise if `pre!(x)` and `post!(x)` are introduced, but do not affect a hypothetical `contract_assert!(x)`.

At the same time, `contract_assert!(x)` is located inside function bodies, and its behaviour can largely be emulated with macros. This is not true of `pre!(x)` and `post!(x)` because macros cannot be placed on function declarations. Supporting assertions on function declarations is one of the primary motivations for language-level contract assertions in the first place, and presumably also a major motivation for introducing an always-enforced flavour.

If `contract_assert!(x)` were nevertheless added. it would also be worth considering its design space overlap with any future support for hygienic macros to C++, which — coincidentally — are spelled *`macroname`*`!` in Rust.

3.3 Mixing regular and always-enforced assertions

Should it be well-formed to mix `pre!(x)` and `pre(y)` on the same function declaration? If so, do we require a guarantee that they be evaluated in lexical order (x before y)?

Consider the following function declaration:

```
void processWidget(Widget* w)
  pre! (w != nullptr)  // critical check, always enforced
  pre (w->isValid());  // not critical, may be ignored or observed
```

If `pre!` is added to the language, it seems reasonable to expect this declaration to be well-formed and for the first precondition assertion to be always evaluated before the second.

However, guaranteeing such behaviour requires always-enforced assertions and regular contract assertions to participate ina single, well-defined evaluation sequence ([P2900R14] Section 3.5.7). While `pre!` *can* be specified in this way, it substantially constrains the design for `pre!`, as discussed in sections 3.5 and 3.6.

3.4 Indirect calls

Should `pre!(x)` and `post!(x)` be checked when a function is called indirectly, such as through a function pointer or from another language?

It seems uncontroversial to require that `pre!(x)` and `post!(x)` be checked when a function is called indirectly through a function pointer:

```
void f(int i) pre!(i > 0);
void (*fptr)(int) = f;
```

```
int main() {
  fptr(-1); // must terminate
}
```

Failing to enforce such checks would undermine the notion of an *always-enforced* check. The same reasoning applies to calls originating from other programming languages that are unaware of C++ contract assertions.

However, meeting this requirement implies that `pre!(x)` and `post!(x)` *must be checked callee-side* for any indirect call. In such cases, the caller does not see the contract assertions, only a function pointer of the appropriate type. However, contract assertions are intentionally not part of the function type, and function pointers themselves cannot carry contract assertions (see [P3327R0]). As a result, it is impossible for the compiler to emit the checks on the caller side.

This requirement therefore imposes additional design constraints that any proposal for always-enforced assertions must respect in order to be both specifiable and implementable.

3.5 Predicate evaluation rules

Should `pre!(x)` follow the same predicate evaluation rules as `pre(x)` — constification, side-effect elision, duplication, and exception handling — or different ones? If so, which?

If `pre!` is designed as a *minimal* extension of the C++26 working draft, it follows that `pre!` should follow the same predicate evaluation rules as `pre`. The only difference between the two would be that `pre!` is constrained to one or two of the four possible evaluation semantics (see Section 3.1). The same conclusion follows if `pre!` is intended as a shorthand for an `always_enforced` label as proposed by [P3400R2], since labels do not alter the meaning of the predicate expression or any aspect of its evaluation.

On the other hand, as discussed in [P3846R0] Concern 1, always-enforced checks represent a fundamentally different feature from build-time configurable contract assertions. They target different use cases, lead to different deployment scenarios, and follow different adoption trajectories. If designed independently — without reference to C++26 contract assertions or to [P3400R2] — they would likely follow different rules, as argued in [P3919R0].

If a check can never be disabled, allowing duplication no longer serves an obvious purpose: If the check fails, duplicated evaluations will not occur; if it succeeds, duplication adds unwanted runtime overhead. For ignorable contract assertions, that overhead was deemed acceptable in exchange for significant implementation and configuration flexibility. For checks that must always be paid for, the trade-off may look very different.

Further, if predicate evaluation occurs exactly once, the side effects of that evaluation become deterministic and therefore less problematic, reducing the need for constification and side-effect elision.

Finally, depending on one's view of the relationship between contracts and program correctness, always-enforced checks may be considered part of deterministic program behaviour rather than "ghost code". Under that interpretation, allowing exceptions to propagate out of predicate evaluation — rather than being intercepted by the contract-violation machinery as is the case for C++26 contract assertions — may be more reasonable.

The characterisation of contract assertions as "ghost code" traces back to a core principle of [P2900R14]: that adding checks to verify program correctness must not alter that correctness (the Contracts Prime Directive). The C++26 rules around duplication, elision, constification, and exception handling all follow from this principle.

Any proposal introducing `pre!` must therefore address whether — and to what extent — the same principle applies to always-enforced contracts. It must clearly specify the chosen predicate evaluation rules, justify them, and consider the implications of these choices for the implementability and ABI compatibility of the feature (see Section 3.6).

If both `pre` and `pre!` exist in the language but follow different predicate evaluation rules, then even when both are evaluated with the same semantic (for example, *enforce*), their predicates do not have the same meaning. Evaluating the same predicate expression `x` could yield different behaviour, introducing a subtle and user-hostile inconsistency — for example, when users initially write `pre!(x)` and later switch to `pre(x)` for performance reasons.

3.6 Implementability and ABI implications

Has this proposal been implemented? Is it implementable? What are the ABI implications of such a proposal?

The design specified in [P2900R14] and adopted into the C++26 working draft has been fully implemented in two major compilers ([P3460R0]). A key principle of this design is that adding `pre(x)` and `post(x)` to a pre-existing function declaration should never break correct code and must be deployable without ABI changes, which would otherwise require toolchain upgrades and significantly delay adoption.

It seems reasonable to hold the newly proposed `pre!(x)` and `post!(x)` to a similarly high standard, particularly if their predicate evaluation rules differ from those of `pre(x)` and `post(x)` (see Section 3.5). Any proposal that introduces such differences must demonstrate implementability and discuss ABI implications.

C++26 contract assertions may be evaluated multiple times, enabling a range of use cases and implementation strategies. The compiler may emit checks callee-side, caller-side, or both. Without an ABI change — such as introducing a distinct function entry point that skips callee-side checks — this can result in duplicate evaluation (see [P3267R0] for a more detailed discussion).

Now, if `pre!(x)` is required to be evaluated exactly once, avoiding an ABI change effectively requires that all checks be emitted *callee-side only*. Otherwise, indirect calls cannot be supported as checks for indirect calls cannot be emitted caller-side (Section 3.4) and emitting the checks twice would not be allowed. Further, supporting mixtures of regular and always-enforced contract assertions on the same declaration (see Section 3.3) would then impose the same callee-side-only restriction on regular contract assertions: caller-side checks would always execute before callee-side ones and thus violate lexical ordering.

However, C++26 contract assertions have been designed from the start to allow emitting the checks both caller-side and callee-side. Restricting them to callee-side-only checks would represent a major design change and preclude many of the use cases they have been designed for — for example, enabling checks in higher-level code for misuse of lower level libraries without recompiling them — which might be impossible for many users — or paying for checks when those libraries call back into themselves.

3.7 Virtual function support

When we add support for pre and post on virtual functions, how should pre! and post! fit into that extension?

Support for `pre` and `post` on virtual functions ([P3097R1]) narrowly missed C++26 but has complete wording reviewed by CWG and is fully implemented in at least one major compiler (GCC). It is therefore expected to land early in the C++29 cycle. Any proposal introducing `pre!` and `post!` should clearly articulate how they fit into that extension.

If `pre!(x)` and `post!(x)` behave identically to `pre(x)` and `post(x)` in all respects except that they cannot be evaluated with *ignore* or *observe* semantic — that is, if they remain subject to constification, side-effect elision, duplication, and are fully forward-compatible with labels — then they are also forward-compatible with virtual function support as specified in [P3097R1]. This is because all our currently proposed post-C++26 extensions for contract assertions — labels [P3400R2], virtual function support [P3097R1], postcondition captures [P3098R1], and user-defined error messages [P3099R1]) — were designed together for full mutual compatibility.

If, however, `pre!` and `post!` differ in their predicate evaluation rules (Section 3.5), additional difficulties arise with virtual functions. In particular, if they are required to be evaluated exactly once, then they can only be checked either on the statically called function or on the final overrider, but not both. This makes it unclear how the default behaviour proposed in [P3097R1] could be implemented. Consider:

```
class Display {
public:
  virtual void post_message(std::string_view s)
    pre (is_ascii(s)) = 0;
};
```

```
class XDisplay : public Display {
public:
  void post_message(std::string_view s) override
    pre (is_utf8(s));
};
```

The abstract base class `Display` requires ASCII strings, while the derived class `XDisplay` can handle UTF-8 (such widening of preconditions is a typical use case of contracts-based programming). Code that uses `XDisplay` directly should be able to display both ASCII and UTF-8 messages without causing a contract violation:

```
int main() {
  XDisplay xDisplay;
  xDisplay.post_message("😈"); // OK, only checks is_utf8
}
```

However, calls made through the `Display` interface must still enforce the stricter precondition, as at compile time it is unknowable at the call site whether the actual dynamic type of display can handle ASCII or not:

```
void boo(Display& display) {
  display.post_message("😈"); // contract violation, is_ascii check fails!
}
```

If `pre` were replaced with `pre!` and required exactly-once evaluation, it is unclear how the above behaviour could be implemented, as this requires emitting checks both caller-side and callee-side.

One remedy suggested in [P3919R0] is automatic inheritance of all precondition and postcondition assertions into derived classes, in which case a check defined in the base class could be emitted callee-side. However, [P3097R1] argues that this approach is not viable in C++. It would introduce false positives (correct code triggering a contract violation if a contract assertion is added), thus violating the Prime Directive of [P2900R14], undermining the utility of the entire contracts facility and preclude many of its intended use cases.

While automatic inheritance has been implemented in other languages — most notably Eiffel — fundamental differences in object model, compilation model, deployment practices, dependency management, and treatment of undefined behaviour mean that the Eiffel approach of auto-inheriting assertions does not translate to C++ (see [P3097R1] for details).

4 Library considerations

4.1 Contract-violation handling library API

Should the library API in header `<contracts>` be extended to allow programmatic distinction between violations of regular and always-enforced assertions? If so, how?

The `contract_violation` object exposes several properties of a violated contract assertion to a user-defined contract-violation handler, including the assertion kind (`pre`, `post`, or `contract_assert`) via the member function `kind()` which returns a value of type `assertion_kind`.

A proposal introducing always-enforced assertions must decide whether the distinction between regular and always-enforced assertions should be reflected in the API. Should `pre!`, `post!`, and/or `contract_assert!` be represented by additional enumerators in `assertion_kind`? If so, how should these enumerators be spelled? Or should there instead be a separate interface, such as a new `is_always_enforced()` function returning a `bool`?

The more general mechanism proposed in [P3400R2] — allowing assertion-control objects to be queried for from the contract-violation handler — is substantially more complex. It must be opt-in, support combined labels, and address behavioural interactions across differently compiled translation units. Adopting such an interface when there is only a single built-in label is questionable, while exposing that label in a bespoke way would create a permanent design inconsistency.

4.2 Relation to standard library hardening

Should hardened preconditions in the C++ standard library be expressible in terms of `pre!(x)`?

C++26 contract assertions have been criticised for the fact that standard library hardening ([P3471R4], also part of the C++26 working draft) cannot be implementable purely in terms of the initial contracts feature set in C++26, despite contract assertions being used as the specification mechanism.

In particular, the C++26 working draft lacks the additional annotations needed to implement a practically deployable hardened standard library. Labels as proposed in [P3400R2] address this gap. In their absence, vendors must rely on vendor-specific attributes (implemented in Clang) or use vendor-specific macros that behave *as-if* they were a `contract_assert` statement (a conforming approach currently used by both libc++ and libstdc++).

This raises the question of whether introducing a syntax for always-enforced assertions could — or should — make standard library hardening implementable purely in terms of that expanded C++26 feature set.

Some have suggested that this may be the intent, even proposing that `pre!(x)` be named “hardened preconditions”. However, hardened preconditions seem fundamentally different from always-enforced assertions: vendors must be able to disable hardened checks when users select a non-hardened build mode, whereas the defining property of `pre!(x)` appears to be that disabling the check is non-conforming.

Acknowledgements

Thanks to Darius Neațu, David Sankel, Ville Voutilainen, J Daniel García Sanchez, John Spicer, Gabriel Dos Reis, Tom Honermann, and Herb Sutter for the discussions that led to this paper. Thanks to Ville Voutilainen for reviewing a draft of this paper and providing valuable feedback.

Bibliography

[N5014] Thomas Köppe: "Working Draft, Programming Languages — C++". 2025-08-05

[P2885R3] Timur Doumler, Gašper Ažman, Joshua Berne, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann: "Requirements for a Contracts syntax". 2023-10–05

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-02-13

[P2961R2] Timur Doumler and Jens Maurer: "A natural syntax for Contracts". 2023-11-08

[P3097R1] Timur Doumler and Joshua Berne: "Contracts for C++: Virtual functions". 2025-12-13

[P3098R1] Timur Doumler, Gašper Ažman, and Joshua Berne: "Contracts for C++: Postcondition captures". 2024-12-11

[P3099R1] Timur Doumler, Peter Bindels. and Joshua Berne: "Contracts for C++: User-defined diagnostic messages". 2025-10-19

[P3267R1] Peter Bindels, Tom Honermann: “C++ contracts implementation strategies”. 2024-05-22

[P3327R0] Timur Doumler: "Contract assertions on function pointers". 2024-10-16

[P3400R2] Joshua Berne: "Controlling Contract-Assertion Properties". 2025-12-15

[P3460R0] Eric Fiselier, Nina Dinka Ranns, and Iain Sandoe: "C++ Contracts: Implementers Report". 2024-10-16

[P3471R4] Konstantin Varlamov and Louis Dionne: "Standard library hardening". 2025-02-14

[P3846R0] Timur Doumler and Joshua Berne: "C++26 Contract Assertions, Reasserted". 2025-10-06

[P3911R0] Darius Neațu, Andrei Alexandrescu, Lucian Radu Teodorescu, and Radu Nichita: "Make Contracts Reliably Non-Ignorable". 2025-11-04

[P3919R0] Ville Voutilainen: "Guaranteed-(quick-)enforced contracts". 2025-12-05

[RO 2-056] NB comment, published in ISO/IEC JTC 1.SC22 N6049: SoV and Collated Comments - ISO/IEC CD 14882 Programming languages — C++, 2025-10-02

[SD-4] "WG21 Practices and Procedures", 2024-12-30