A proposed plan for extending Contracts in C++29

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.com)

Document #: P3850R0
Date: 2026-05-12
Project: Programming Language C++
Audience: EWG, LEWG

Abstract

C++26 contract assertions are, by design, a minimal viable product. In order to make the feature more usable at scale and across a wider variety of scenarios, key extensions are needed and are needed soon. In this paper, we propose a roadmap for the essential extensions that are high-impact, already relatively mature and understood, address many of the concerns that were raised during the standardisation of the C++26 feature set, and will make C++29 contract assertions a more complete and powerful tool.

Contents

1 Motivation 2
2 The plan 2
2.1 Virtual functions 4
2.2 Library API for triggering contract-violations 4
2.3 Labels 5
2.4 Implicit contract assertions 6
2.5 Postcondition captures 6
2.6 User-defined messages 7
3 Extensions not included in the plan 7
Acknowledgements 8
Bibliography 8

1 Motivation

At the March 2026 WG21 meeting in Croydon, the ISO C++ committee completed the C++26 Standard, including the addition of *contract assertions*. This milestone is the result of a sustained effort by the authors of [P2900R14] and the WG21 subgroups responsible for evaluating the proposal: SG21, EWG, and CWG.

This success was made possible in part by the existence of a clear roadmap [P2695R1] for the C++26 cycle. That roadmap was adopted by SG21 with strong consensus in 2022 and was followed throughout the entire standardisation process.

The feature set included in C++26 provides a solid foundation, but lacks a number of extensions needed to support additional use cases for which there is already a demonstrated need. In particular, further work is required to make contract assertions truly usable at scale: in very large codebases, across libraries developed and distributed independently, and in specialised environments such as safety-critical systems.

This is by design. WG21 has a long-established practice to standardise powerful new language features *incrementally*, and this approach has proven successful in the past, for example in the evolution of `constexpr`. Our goal for C++29 is to extend the C++26 contract-assertion facility in ways that unlock these additional use cases.

Our other goal for C++29 is to address most, if not all, concerns that were raised repeatedly during the C++26 standardisation phase.[1] We remain committed to addressing those concerns to the fullest extent possible while preserving the intended design and functionality of the C++26 contract-assertion facility, and we thus recommend prioritising proposals that accomplish this.

Given the success of the roadmap-driven approach in the previous cycle, we believe that the best path forward is to establish a corresponding roadmap for C++29. This paper proposes such a roadmap. It is composed of independent proposals for extensions building upon the foundation we shipped in C++26. In this paper, we identify the extensions that we believe should be prioritised during this cycle, and seek commitment of EWG — the group now responsible for this work, following the dissolution of SG21 — to that roadmap.

2 The plan

The design space for extending contract assertions toward a more complete facility is large (see [P2755R1]). It is neither realistic nor desirable to attempt to standardise all these extensions within the C++29 timeframe. Some remain at an early stage of design, committee time is necessarily limited, and the features proposed for C++29 should not result in unrealistic implementation burdens.

[1] For concerns raised during the C++26 standardisation phase, see [P3173R0], [P3478R0], [P3506R0], [P3573R0], [P3829R0], [P3835R0], [P3849R0], [P3851R0], [P3911R2], [P4044R0]; for responses, see [P3500R1], [P3591R0], [P3846R0], [P3912R0], [P3946R0].

We therefore propose to focus on a concrete subset of extensions. In particular, priority should be given to proposals that satisfy the following criteria:

A. **Value:** Unlocks important use cases, provides significant value to a diverse set of C++ users.
B. **Urgency:** Addresses major concerns raised during standardisation of the C++26 feature set.
C. **Maturity:** The associated proposal is design-complete and ready for EWG/LEWG review.
D. **Review:** The proposal has already undergone one or more rounds of review in a WG21 subgroup (SG21 and/or EWG) and feedback has been incorporated.
E. **Specification:** The proposal is accompanied by complete wording, with no known unresolved specification questions.
F. **Implementation:** The proposal has at least one complete and working implementation in a major compiler.

The following six extensions are, in our view, the best candidates to move forward in the near term. All six extensions satisfy three or more of the criteria listed above. In particular, we consider all six to be high-value, and all six have proposals ready for EWG/EWG review. We therefore recommend them for inclusion in C++29, in the following order of priority:

Feature	**Proposal**	**Audience**	**Criteria satisfied**	**Scope**
Support for `pre` and `post` on **virtual functions**	[P3097R2]	EWG	A, B, C, D, E, F	medium
Library API for **explicitly triggering contract violations**	[P3290R4]	LEWG	A, B, C, D, E	small
Labels for controlling contract-evaluation semantics in code	[P3400R3]	EWG + LEWG	A, B, C, D	large
Implicit contract assertions guarding against core-language UB	[P3100R6]	EWG + LEWG	A, C, D, E	large
Postcondition captures	[P3098R2]	EWG + LEWG	A, C, D	medium
User-defined messages on contract assertions	[P3099R2]	EWG + LEWG	A, C, E	small

Unlike with the roadmap [P2695R1] that led to C++26 contracts, these extensions are independent of one another and can all follow the normal process of being seen, reviewed, and forwarded by EWG/LEWG and then CWG/LWG.
Further, unlike with [P2695R1], SG21 is no longer part of the WG21 pipeline as it has been dissolved; the deadlines for this process are thus given by [P1000R7]. Considering the relative maturity of the proposed extensions, we believe it is possible to approve them for the C++29 working draft much sooner — within the next few WG21 meetings.

In the following subsections, we provide a brief summary of each extension.

2.1 Virtual functions

Runtime polymorphism is a fundamental part of C++. Many libraries and APIs are designed around virtual functions. Yet, C++26 contract assertions do not integrate with them: placing `pre` or `post` on the declaration of a virtual function results in the program being ill-formed. This is surprising and hostile to users, and prevents many from using contract assertions effectively in their codebases:

```
class QIODevice {
public:
  // ...
  virtual qint64 readData(char* data, qint64 maxSize)
    pre (data != nullptr) = 0;  // should work, but doesn't in C++26
};
```

In [P3097R2], we have a mature and scalable design for adding this functionality. It is easy to understand and teach, has complete wording, and has a working implementation in GCC. Remarkably, this design only narrowly missed inclusion in C++26: it previously gained strong consensus in EWG, was merged into the main C++26 contracts proposal, and underwent thorough wording review in CWG. However, at the last minute, EWG decided to remove it again due to concerns raised in [P3506R0] and [P3573R0] that our approach to contract assertions on virtual functions diverges from the canonical model in Eiffel and D, and thus lacks sufficient deployment and usage experience.

To help re-establish consensus, in [P3600R0] we have conducted the most comprehensive analysis to date of the use cases, available design space, and trade-offs among the many possible designs for contract assertions on virtual functions. Based on this extensive work, we are more confident than ever that the Eiffel/D model is a poor fit for C++, and that the design proposed in [P3097R2] is the right solution. We recommend that EWG re-review and re-approve this proposal at the next WG21 meeting to close this significant gap in the C++ contract-assertion facility.

2.2 Library API for triggering contract-violations

In [P3290R4], we propose adding library functions `handle_observed_contract_violation`, `handle_enforced_contract_violation`, and `handle_quick_enforced_contract_violation` to directly trigger a contract violation with the desired evaluation

semantic from anywhere in user-defined code. This small, straightforward, library-only proposal allows users to hook into the standard contract-violation mechanism without having to use contract assertions, addressing many of the concerns raised about the C++26 design.

Users who, for whatever reason, need to preserve the semantics of their existing contract-assertion macros — such as no constification, no elision or duplication, and no conversion of exceptions thrown during predicate evaluation into contract violations — can continue to use their preferred semantics while still benefiting from using the standard global contract-violation handler.

For example, with [P3290R4] a user-defined, contracts-integrated assertion macro with no constification, "evaluated exactly once when checked" semantics, and exception pass-through may be defined as follows:

```
#if ENABLE_MY_ASSERT
  #define MY_ASSERT(expr, msg) \
    if (!(expr)) handle_enforced_contract_violation(msg);
#else
  #define MY_ASSERT(expr, msg)
#endif
```

2.3 Labels

Labels, proposed in [P3400R3], are a major extension to C++26 contracts that make the feature significantly more scalable and unlock many important use cases.

While [P3400R3] proposes many kinds of labels, for C++29 we should focus on those that allow users to control and constrain the evaluation semantics of contract assertions (and sets of contract assertions) directly in code. Such labels directly address a number of important use cases, including:

- Contract assertions that are *always* enforced (e.g., for security-critical checks) can be defined with a label that restricts the semantics to *enforce* and *quick-enforce* only.
- Contract assertions that are *never* enforced (e.g., those newly added to a legacy codebase and not yet trusted) can be defined with a label that restricts the semantics to *ignore* and *observe* only.
- Contract assertions that always have a single evaluation semantic — and are thus fully deterministic, with no implementation-defined runtime behaviour (e.g. for unit-testing a contract-violation handler) — can be defined with an appropriate label.
- Contract assertions that belong to a particular unit, such as a given library, can be grouped by adding a label `my_library`, allowing their evaluation semantics to be controlled as a group and independently from other assertions.
- Contract assertions that have particular characteristics such as being particularly expensive to evaluate can be labelled `audit` and enabled only in specialised builds.

The first use case — always-enforced assertions — still has some unresolved questions regarding ABI impact (see [P3912R0]) that were not resolved in the C++26 timeframe. Given

that multiple NBs have identified this as an important use case ([P3911R2], [P3919R0]), we should make a concentrated effort to resolve these questions in the C++29 timeframe.

[P3400R3] provides the necessary syntax — `pre<label> (expression)` — as well as the language and standard library infrastructure required to programmatically define the semantics of each label in user code and to combine labels:

```
T* binarySearch(T* begin, T* end, const T& value)
  pre <my_library | audit> (is_sorted(begin, end));
```

2.4 Implicit contract assertions

Implicit contract assertions, proposed in [P3100R6], are a key part of a broader, holistic strategy to detect, mitigate, and ultimately eliminate undefined behaviour in existing C++ code. They introduce runtime checks guarding against core-language undefined behaviour that behave like the *explicit* contract assertions in C++26 (`pre`, `post`, and `contract_assert`). As such, they can be ignored, observed, enforced, quick-enforced, and integrated with the global contract-violation handler. However, unlike explicit contract assertions, they are *implicitly* inserted by the compiler to guard against conditions like out-of-bounds access into an array or signed integer overflow.

These implicit checks can be enabled by users at build time, with no code changes needed, similar to existing compiler options such as Clang's `-fbounds-safety` or GCC's `-ftrapv`, but integrated into a standardised framework.

This proposal has already been reviewed by SG21, SG23, and twice by EWG, gaining strong consensus to pursue further on all four occasions. The primary missing piece at the last EWG review was complete wording for all ~80 cases of explicit core-language undefined behaviour addressed by the proposal. We have now provided such wording in the latest revision.

2.5 Postcondition captures

Postcondition captures, proposed in [P3098R2], address several limitations of C++26 contract assertions at once.

First, C++26 provides no way for a postcondition assertion to refer back to the program state at the time the function was called (e.g., the "old" value of a parameter). This makes common postconditions — such as stating that `push_back` increments the container size by one when returning normally — inexpressible in C++26. Postcondition captures provide a familiar and intuitive syntax for capturing such values:

```
void push_back(T& item)
  post [old_size = size()] (size() == old_size + 1);
```

Additionally, in C++26 it is impossible to refer to by-value parameters in postcondition assertions unless the parameter is declared `const`:

```
int f(int i)
  post (r: r >= i); // error in C++26: `i` is not const
```

This is awkward and unintuitive, as users have long been taught that such `const` qualifiers on non-defining declarations have no observable effect. The situation is even worse for coroutines, where referring to by-value parameters in postconditions is not possible at all, because declaring a parameter `const` would prevent the coroutine machinery from moving that parameter into the coroutine frame and thus render the program ill-formed.

These limitations arise because, in some situations, the user needs to refer to the original parameter value passed to the function and not the actual parameter object. While the `const` qualifier is a workaround to ensure these two values cannot diverge, postcondition captures allow the user to capture the initial value without imposing any additional requirements on how the parameter is used by the function:

```
int f(int i)
  post [i] (r: r >= i);      // OK, capture parameter by copy
```

2.6 User-defined messages

This small but useful and frequently requested extension allows users to specify a custom diagnostic message for a contract assertion. Clang already supports such custom messages via a vendor-specific attribute. [P3099R2] proposes standardising this capability for C++29. The proposed syntax is exactly what most users would expect:

```
T& operator[] (size_t i)
  pre (i < size(), "Out-of-bounds access");
```

This feature is straightforward to specify and implement, and the above syntax is consistent with existing facilities such as `static_assert`.

3 Extensions not included in the plan

A number of other possible extensions to C++26 contract assertions have either significant unresolved design issues, relatively lower impact than those included in the plan, or both. While progress on these extensions should continue during the C++29 cycle, we feel that it is too early to commit to any particular shipping vehicle for them. These extensions include:

- `pre` and `post` on function pointers ([P3271R1], [P3327R0])
- Class invariants ([P2755R1] Sections 2.2.11, 3.3.2)
- Labels for purposes other than controlling evaluation semantics, including:
  - local contract-violation handlers ([P3400R3] Section 3.3.4)
  - message/comment manipulation ([P3400R3] Section 3.3.6)
  - side channel for passing user data to the `contract_violation` object ([P3400R3] Section 3.3.7)
- Procedural function interfaces ([P0465R0], [P2755R1] Sections 2.2.7, 2.2.8, 3.3.1)

- Declaring entities used by only contract assertions ([P2755R1] Section 2.2.20, 3.4.4)
- Postcondition return value destructuring ([P2755R1] Sections 2.2.10, 3.4.2)
- Contract assertion `requires` clauses ([P2755R1] Sections 2.2.3, 3.4.3)
- Pack expansion of contract assertions ([P2755R1] Sections 3.6.6)
- Precondition checking operator ([P2755R1] Section 3.6.9)
- `pre` and `post` on trivial functions ([P2755R1] Section 3.6.5)
- Ambient control of labels on explicit and implicit contract assertions ([P3400R3] Section 4.2)
- Postconditions on values `co_yielded` / `co_returned` from coroutines

Acknowledgements

Thanks to Lucian Radu Teodorescu, Erich Keane, Hana Dusíková, Nevin Liber, Peter Bindels, Oliver Rosten, Iain Sandoe, Gašper Ažman, and Rostislav Khlebnikov for reading a draft of this paper and providing valuable feedback.

Bibliography

[P0465R0] Lisa Lippincott: "Procedural function interfaces". 2016-10-16

[P1000R7] Herb Sutter: "Proposed C++ IS schedule". 2026-01-13

[P2695R1] Timur Doumler and John Spicer: “A proposed plan for contracts in C++”. 2023-02-09

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos: "A Bold Plan for a Complete Contracts Facility". 2024-04-11

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

[P3097R2] Timur Doumler and Joshua Berne: "Contracts for C++: Virtual functions". 2026-05-11

[P3098R2] Timur Doumler, Gašper Ažman, and Joshua Berne: "Contracts for C++: Postcondition captures". 2026-05-12

[P3099R2] Timur Doumler, Peter Bindels. and Joshua Berne: "Contracts for C++: User-defined diagnostic messages". 2026-05-11

[P3100R6] Timur Doumler and Joshua Berne: "A framework for systematically addressing undefined behaviour in the C++ Standard". 2026-05-12

[P3271R1] Lisa Lippincott: "Function Types with Usage (Contracts for Function Pointers)". 2024-10-04

[P3290R4] Joshua Berne, Timur Doumler, and John Lakos: "Integrating Existing Assertions with Contracts'. 2026-05-02

[P3400R3] Joshua Berne: "Controlling Contract-Assertion Properties". 2026-05-11

[P3500R1] Timur Doumler, Gašper Ažman, Joshua Berne, and Ryan McDougall: "Are Contracts "safe"?". 2025-02-09

[P3506R0] Gabriel Dos Reis: "P2900 Is Still not Ready for C++26". 2024-11-19

[P3573R0] Michael Hava, J. Daniel Garcia Sanchez, Ran Regev, Gabriel Dos Reis, John Spicer, Bjarne Stroustrup, J.C. van Winkel, David Vandevoorde, Ville Voutilainen: "Contracts concerns". 2025-01-12

[P3591R0] Joshua Berne and Timur Doumler: "Contextualizing Contracts Concerns". 2025-02-03

[P3600R0] Timur Doumler: "Contract assertions on virtual functions: a principled design approach". 2026-07-15

[P3829R0] David Chisnall, John Spicer, Gabriel Dos Reis, Ville Voutilainen, and Jose Daniel Garcia Sanchez: "Contracts do not belong in the language". 2025-09-02

[P3835R0] John Spicer, Ville Voutilainen, Jose Daniel Garcia Sanchez: "Contracts make C++ less safe — full stop!". 2025-09-03

[P3846R1] Timur Doumler and Joshua Berne: "C++26 Contract Assertions, Reasserted". 2025-11-03

[P3849R0] Harald Achitz: "SIS/TK611 considerations on Contract Assertions". 2025-09-27

[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". 2025-09-29

[P3911R2] Darius Neațu, Andrei Alexandrescu, Lucian Radu Teodorescu, Radu Nichita, and Herb Sutter: "Make Contracts Reliably Non-Ignorable". 2026-01-14

[P3912R0] Timur Doumler, Joshua Berne, Gašper Ažman, Oliver Rosten, Lisa Lippincott, and Peter Bindels: "Design considerations for always-enforced contract assertions". 2025-12-15

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

[P3946R0] Andrzej Krzemieński: "Designing enforced assertions". 2025-12-14

[P4044R0] Lucian Radu Teodorescu: "Just `pre!`. Mandatory precondition for contracts". 2026-03-08