Exploring strict contract predicates

Timur Doumler (papers@timur.audio)
Lisa Lippincott (lisa.e.lippincott@gmail.com)
Joshua Berne (jberne4@bloomberg.net)

Document #:	P3499R1
Date:	2025-02-09
Project:	Programming Language C++
Audience:	EWG

Abstract

The lack of support for so-called *strict contracts* — contract assertions that cannot have side effects or undefined behaviour when evaluated — is the subject of sustained opposition to [P2900R13], the Contracts facility proposed for C++26. In this paper, we explore an actionable — i.e., specifiable and implementable — design for such a feature.

1 Context

It has been suggested in [P3173R0] and [P3362R0] that the Contracts facility as proposed in [P2900R13] is not fit for standardisation unless it supports so-called *strict* contracts, i.e., contract assertions that cannot have side effects or undefined behaviour when evaluated, as proposed in [P2680R1] and [P3285R0]. Over the last three years, this idea has been discussed and polled multiple times in SG21, SG23, and EWG; all of these groups consistently had consensus against pursuing it (for a detailed history, see [P2899R0] Section 3.6.1). The most recent of these polls took place in EWG in November 2024:

EWG Poll, Wrocław, 2024-11-19

As suggested in P3362R0 / P2680 / P3285, the contracts proposal in P2900's Minimal Viable Product shall be changed to incorporate stricter contracts in addition to regular contracts.

SF	F	N	A	SA
10	6	3	14	16

Result: Consensus against, but P2900 would be in danger of failure in plenary

EWG and LEWG have both approved the design of [P2900R13], without strict contracts, and forwarded it to wording review for inclusion in the C++ Standard. However, as indicated by the poll above, and stated in the more recent papers [P3506R0] and [P3573R0], the issue of strict contracts remains a subject of sustained opposition against inclusion of [P2900R13] in the C++ Standard.

To address the concerns in [P3506R0] and [P3573R0], in this paper we explore how a possible design for strict contracts could actually be specified, and what that would look like in practice.

2 The problem

Strict contracts is an approach that restricts the predicates of contract assertions to expressions that can be statically proven to not introduce undefined behaviour (excluding data races) and to not have side effects outside of its cone of evaluation; predicates for which such a proof cannot be constructed are ill-formed. When undefined behaviour cannot be prevented statically (e.g., integer overflow), it is instead redefined to be some well-defined behaviour at runtime.

This approach necessarily constrains the set of expressions that are allowed in the predicates of strict contracts. For example, in a strict predicate it is not possible to call any function that itself has not been statically proven to satisfy the required properties.

Such strict predicates are too restrictive for general use as a runtime contract checking facility. Therefore, [P2680R1] and [P3285R0] also allow the user to write predicates that do not satisfy the constraints of strict contracts, called *relaxed contracts*. The semantics of relaxed contracts are equivalent to the semantics of contract assertions as proposed in [P2900R13].

Unfortunately, the proponents of strict contracts have not produced a complete specification, and it appears doubtful whether the approach is indeed specifiable and implementable.

Deeper analysis of the idea in [P3376R0] and [P3386R0] revealed that strict contracts, based on the principles that can be gleaned from [P3285R0], result in only a very small number of predicates that would be viable to express, with a huge amount of new language complexity needed to achieve anything beyond the most basic arithmetic operations.

In particular, we do not yet know whether or how one could use pointers and references to objects, and call member functions, in a strict contract predicate, as we are not aware of any technique applicable to C++ that could prove that a pointer or reference points to a valid object.

It seems possible or even likely that no approach using *local* static analysis, such as the sketch for a `std::object_address` facility provided in [P3285R0], could ever provide such a proof, and that only the introduction of *global* language constraints on having mutable references to objects can achieve this, such as a borrow checker ([P3390R0]), mutable value semantics ([Racordon2022]), or outlawing mutation of objects altogether like in pure functional languages; see also [Baxter2024].

3 The approach

3.1 Set of allowed expressions

Given the fundamental problem described above, strict contracts in today's C++ can necessarily express only a very limited set of predicates; nevertheless, in order to make any progress on this issue at all, it is useful to try and concretely define such a set.

As a starting point, we can specify strict contracts to only allow predicates for which we are confident that absence of undefined behaviour (excluding data races) can indeed be accomplished with the technology of today. Initially, this will be a very small set, but it provides an evolutionary path towards further expansion. To this end, we have identified the following set of expressions:

— literals of arithmetic or enumeration type,

— *id-expression*s that denote a non-`volatile` variable with arithmetic or enumeration type (notably, this excludes pointers and references),

— unary-expressions where the operator is one of the following: `+`, `-`, `!`,

— *binary-expression*s where the operator is one of the following: `+`, `-`, `/`, `%`, `*`, `!`, `,`, `^`, `|`, `||`, `&`, `&&`, `<<`, `>>`,

— *conditional-expression*s where the operator is `?!`,

— *relational-expression*s where the operator is `<`, `>`, `<=`, `>=`,

— *equality-expression*s where the operator is `==`, `!=`,

— *compare-expression*s where the operator is `<=>`,

— core constant expressions of arithmetic or enumeration type.

If we restrict strict predicates to just the above expressions, and make all other expressions in strict predicates ill-formed, we exclude modifications of *any* variables and thus by extensions exclude any side effects outside of the cone of evaluation of the predicate.

Further, we can enumerate all instances of undefined behaviour that can occur when evaluating such a predicate according to the C++ Standard today (see [P1705R1]), excluding data races:

— Signed integer overflow or underflow,

— Converting a floating point value to a type that cannot represent that value,

— Division by zero,

— Shifting by a negative amount,

— Shifting by equal or greater than the bit-width of the type.

3.2 Redefining undefined behaviour

The next step is to redefine the above instances undefined behaviour to instead be well-defined behaviour when encountered during the evaluation of a strict contract predicate. We see three possible options for this:

1. Specify a concrete value that such an operation should produce, for example wraparound or saturation arithmetics for integer overflow;
2. Specify that the operation should produce an unspecified *erroneous* value;
3. Specify that the behaviour is a contract violation; if the contract-violation handler is called, the value returned by `kind()` is `implicit`, and the value returned by `detection_mode()` is a newly introduced enumeration value `arithmetic_error`.

As discussed in more detail in [P3386R0], the first approach is actively user-hostile as it leads to masking of bugs and false negatives. The second approach seems viable but suboptimal because it does not tie in with the rest of the Contracts facility in case a defect in the predicate is detected.

We therefore recommend the third approach as it allows for more fine-grained contract-violation handling and is fully consistent with the future direction towards a safer C++ laid out in [P3100R1] and [P3229R0].

3.3 Syntax

Finally, we need to syntactically distinguish strict and relaxed contract assertions. The options are:

1. Specify the default syntax `pre(x)` to denote a relaxed contract, and require an additional syntactic qualifier such as `pre strict(x)` for strict contracts;
2. Specify the default syntax `pre(x)` to denote a strict contract and require an additional syntactic qualifier such as `pre relaxed(x)` for relaxed contracts;
3. Make the default syntax ill-formed and require an additional syntactic qualifier for both strict and relaxed contracts.

Anything other than the first option would be a breaking change to [P2900R13]. On the other hand, [P2680R1] argues for the second option. This design question was discussed and polled by EWG at the November 2024 WG21 meeting in Wrocław. EWG decided against both the second and third option:

EWG Poll, Wrocław, 2024-11-19

Change P2900 to either make `strict` the default behavior, or to force opt-in (no default).

SF	F	N	A	SA
6	7	9	20	7

Result: Consensus against

This leaves the first option (the default syntax should denote a relaxed contract, as it does in [P2900R13] today) as the only remaining possibility. By choosing this option, we enable strict contracts to be added to [P2900R13] as a non-breaking extension at a later time.

3.4 Limitations

While strict contracts as described above seem specifiable and implementable, it is worth highlighting that they are severely limited. Remember that in such strict predicates:

— Any operation on values other than of built-in arithmetic or enumeration type is ill-formed;

— Dereferencing any pointers is ill-formed;

— Using any references to objects is ill-formed;

— Calling any existing member function on any object is ill-formed.

For example, we could not even check the size of a `std::vector` in a strict predicate, or whether it is empty, not even if that `std::vector` is passed in by value, because we cannot construct a proof that the `this` pointer is valid. It seems therefore that strict predicates are of little practical use for adding contract assertions to any kind of real-world C++ codebase.

4 Summary

It is indeed possible to construct a subset of expressions in today's C++ for which we can exclude the possibility of side effects outside of the cone of their evaluation and eliminate all undefined behaviour, and define *strict contracts* as contract assertions whose predicates are restricted to that subset.

However, an attempt to actually construct a concrete such subset without major reinventions such as the introduction of a Rust-like borrow checker to C++ leaves us with a set of expressions so severely limited that it seems to be of no practical use. Therefore, we remain unconvinced that strict contracts are an approach worth pursuing, and instead recommend the direction proposed in [P3100R1].

We are certainly not opposed to exploring strict contracts further; however, given that EWG decided that the default contracts syntax should denote *relaxed contracts* as proposed in [P2900R13], and strict contracts should use an opt-in syntax, strict contracts can be added to [P2900R13] as a non-breaking extension at a later time, and there is no reason why [P2900R13] should be blocked by them.

In any case, we hope that the above description of what an actionable specification of such a feature would actually look like in practice will be helpful towards increasing consensus on shipping the initial Contracts facility proposed in [P2900R13] in C++26.

Revision History

R0 → R1:

— Fixed an incorrect poll citation

— Editorial changes

Bibliography

[Baxter2024] Sean Baxter. Why Safety Profiles Failed. https://www.circle-lang.org/draft-profiles.html, 2024-10-24.

[P1705R1] Shafik Yaghmour. Enumerating Core Undefined Behavior. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html, 2019-09-28.

[P2680R1] Gabriel Dos Reis. Contracts for C++: Prioritizing Safety. https://wg21.link/p2680r1, 2022-12-15.

[P2899R0] Joshua Berne, Timur Doumler, Rostislav Khlebnikov, and Andrzej Krzemieński. Contracts for C++ — Rationale. https://wg21.link/p2899r0, 2025-01-13.

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r13, 2025-01-13.

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne. Undefined and erroneous behaviour is a contract violation. https://wg21.link/p3100r1, 2024-10-16.

[P3173R0] Gabriel Dos Reis. P2900R6 May Be Minimal, but It Is Not Viable. https://wg21.link/p3173r0, 2024-02-15.

[P3229R0] Timur Doumler and Joshua Berne. Making erroneous behaviour consistent with Contracts. https://wg21.link/p3229r0, 2025-01-13.

[P3285R0] Gabriel Dos Reis. Contracts: Protecting The Protector. https://wg21.link/p3285r0, 2024-05-15.

[P3362R0] Ville Voutilainen and Richard Corden. Static analysis and 'safety' of Contracts, P2900 vs. P2680/P3285. https://wg21.link/p3362r0, 2024-08-11.

[P3376R0] Andrzej Krzemieński. Contract assertions versus static analysis and 'safety'. https://wg21.link/p3376r0, 2024-10-14.

[P3386R0] Joshua Berne. Static Analysis of Contracts with P2900. https://wg21.link/p3386r0, 2024-10-15.

[P3390R0] Sean Baxter and Christian Mazakas. Safe C++. https://wg21.link/p3390r0, 2024-09-11.

[P3506R0] Gabriel Dos Reis. P2900 Is Still Not Ready for C++26. https://wg21.link/p3506r0, 2024-11-19.

[P3573R0] Michael Hava, J. Daniel García Sanchez, Ran Regev, Gabriel Dos Reis, John Spicer, Bjarne Stroustrup, J.C. van Winkel, and Daveed Vandevoorde. Contract concerns. https://wg21.link/p3573r0, 2025-01-12.

[Racordon2022] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation Strategies for Mutable Value Semantics. https://www.jot.fm/issues/issue_2022_02/article2.pdf, 2022-02.