Contracts for C++ — Rationale

Document #:	P2899R1
Date:	2025-03-14
Project:	Programming Language C++
Audience:	SG21 (Contracts), EWG, LEWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <mail@timur.audio>
	Rostislav Khlebnikov <rkhlebnikov@bloomberg.net>
	Andrzej Krzemieński <akrzemi1@gmail.com>

Abstract

SG21 has, throughout the C++26 cycle, meticulously followed the plan set forth in [P2695R1] to achieve the goal of delivering a minimal viable product (MVP) for a Contracts feature in C++26. That product is [P2900R14] — Contracts for C++. This paper is a companion paper to [P2900R14] and aims to provide the history, the reasoning, and some of the motivation behind the decisions made for that proposal.

Contents

Revision History

Revision 1

- Updated this paper to coordinate with [P2900R14]
- Added the "Implementation-Defined Behavior" section
- Made various clarifications and corrections and included additional references

Revision 0

- First release of this paper, relative to [P2900R13]

1 Introduction

The Contracts[1] paper, [P2900R14], introduces the basic terminology of the proposed Contracts feature, describes its scope and design, and contains the corresponding proposed wording changes. The paper contains few details on the motivation for the design decisions in that proposal, the discussion of alternative design choices that have been considered, and the history of how the proposal and all its parts came about, yet this information is important context. This companion paper provides those details, which will help explain [P2900R14], especially for anyone seeking to contribute further productive work building on what has been done so far. Other supplementary material for [P2900R14] can also be found here.

In this paper, you will find three primary sections. "Overview" describes the general motivation for the Contracts proposal [P2900R14], the goals we wish to achieve with the feature set of this proposal, and an overview of relevant prior art in other programming languages and the history of standardizing a Contracts feature for C++. "Proposed Design" contains, for each corresponding subsection in [P2900R14], a summary of the motivation for the proposed design, a history — as complete as possible — for the design decisions in that section, and citations to the many papers, written by members of WG21 and SG21, that have contributed to this design. Finally, "Errata" contains corrections for errors in the frontmatter of [P2900R14] that cannot be corrected directly in that paper because it has already been approved by WG21 plenary and, therefore, cannot be revised further.

When a particular paper achieved consensus in SG21 and resulted in that paper's proposed changes being adopted into [P2900R14] or earlier revisions that preceded it, we will largely defer to that paper for a thorough discussion of the motivation and use cases and a deeper understanding of the intent that led to the current state of our proposal. When these design decisions were discussed and made in SG21 or other relevant WG21 subgroups, we summarize those here, along with polls that were taken and their results as called by the chair at the time.

Throughout this paper, we will consistently use the terminology established in [P2900R14], Section 2 to refer to Contracts-related concepts and entities, even though the many papers and other Contracts-related works that we will be citing might be using differing terminology for the same concepts. Such older terminology will be called out whenever necessary for understanding.

2 Overview

2.1 What Are Contracts For?

Section 2.1 of [P2900R14] contains a formal description of what contracts are, explains the crucial distinction between *plain-language contracts* and *contract assertions*, defines a *Contracts facility* as a language feature that lets the user write such contract assertions, and lists the purpose of different contract assertion *kinds* such as precondition and postcondition assertions. What Section 2.1 of [P2900R14] does *not* do is answer the crucial question of what the proposed Contracts facility is actually *for*.

[1] We capitalize "Contracts" or we explicitly say "the Contracts feature" or "the Contracts facility" when we are referring to the feature, and we use lowercase "contracts" when we are referring to contracts themselves (as in "plain-language contracts"). We apply the same convention to Ranges, Profiles, Coroutines, and Concepts.

The primary purpose of the Contracts facility proposed in [P2900R14] is to allow the programmer to express expectations about program correctness at certain points in a C++ program, particularly when calling and returning from functions; optionally verify during program evaluation that those expectations are met to thus *identify program defects*; and flexibly, portably, and scalably manage how identified defects should be mitigated.

The existing `assert` macro in header `<cassert>` is arguably also a Contracts facility, albeit a rudimentary one. It offers no contract-violation handling strategy other than calling `std::abort()` nor any kind of customization of its behavior other than transforming into an empty expression by defining `NDEBUG`. For this reason, many codebases resort to custom assertion macros that offer more options. However, these macros have no language support and thus suffer from a number of limitations. An alternative way of stating the goal of [P2900R14] is that the proposed Contracts facility is meant as a superior alternative to the usage of C `assert` and similar *assertion macros* by removing those limitations.

- Not relying on *macros* to express assertions removes many associated problems such as parsing issues,[2], ODR issues,[3] the risk of bit rot,[4] and the invisibility of *ignored* assertions to tooling due to the macros being textually removed.
- Being able to place precondition and postcondition assertions on the *declaration* of a function, rather than inside the function body, makes them visible without having to inspect the function definition and thus makes them more utilizable for humans, compilers, and tooling, such as IDEs and static analysis.
- Providing a standard syntax that is shared across all components of a program — and eventually the entire C++ ecosystem — makes writing, reading, and processing contract assertions portable and scalable and is a vast improvement over the status quo in which every library is using their own assertion macros.
- Having a program-wide *contract-violation handler* that can be defined by the user and supplied at *link time* makes the *handling* of contract violations at run time equally portable and scalable.
- Being able to directly refer to the *return object* of a function makes many postconditions expressible that cannot be expressed noninvasively with macros.
- Providing a flexible system of *evaluation semantics* that can be chosen in many different ways enables many more use cases and tools than a system where contract checks are either turned on or turned off with a single per-translation-unit build flag.
- Having a design based on the Contracts Prime Directive ([P2900R14]'s Design Principle 1) means that, unlike with macro-based facilities, adding a contract assertion to a program or

[2] Macros are inherited from C and are not aware of C++ specific syntax such as braced initialization or template argument lists, leading to parsing errors when the predicate contains expressions like `Widget{1, 2}` or `array<int, 3>`.

[3] When an inline function in a shared header contains an assertion macro, compiling one translation unit with macros turned on and another translation unit with macros turned off in the same function and then linking those together results in an ODR violation.

[4] When assertion macros are turned off, they are textually removed by the preprocessor, which means that the program will still compile if the predicate contains an undeclared identifier or otherwise broken code — until the macros are eventually turned on again.

enabling the checking of such an assertion should not alter the correctness of that program, thereby preventing "Heisenbugs" where a bug appears or disappears when the developer attempts to track it down using assertions.

When designing a Contracts feature for C++, the main challenge is the large number of different use cases for such a facility, especially in a multiparadigm language such as C++. Any choice to prioritize a particular use case can lead to a particular design that might be less ideal for satisfying some other use case.

This property of Contracts is what makes achieving consensus on a particular design for this feature so difficult. It is also a likely explanation for why, after over twenty years of attempts to standardize Contracts for C++, no such proposal has yet been adopted into the C++ Standard.

When SG21 started its work, i.e., after C++20 was feature-complete and failed to include a Contracts facility, we first focused on identifying use cases for Contracts and determining their priority. [P1995R1] lists 197 different use cases which have been categorized by thirty participants into the three priority levels *Must Have*, *Nice to Have*, and *Not Important*. Follow-up papers for this in-depth study include [P2185R0], which categorizes these use cases into categories such as *Correctness* and *Optimization*, and [P2032R0], which analyzes how well pre-SG21 Contracts proposals satisfy those use cases.

While some of the proposed use cases are very specific, we can identify five broad categories of needs into which all known use cases fall, and we list them here in order of priority.

1. **Documentation**: Express the contracts of functions and other software components in *code*, as opposed to comments or separate documentation, thus making them consumable by human readers, IDEs, and other tooling and facilitating correct use of interfaces while developing a program.
2. **Runtime checking**: Evaluate contract predicates *during program execution* to portably and scalably identify defects.
3. **Static analysis**: Provide contract assertions as additional input for static analysis tools to identify defects from the source code without running the program, e.g., through symbolic evaluation.
4. **Optimization**: Provide contract assertions as input for optimizers that can *assume* the contract predicates to be true, and optimize the program based on this assumption to thus generate faster code.
5. **Verification**: Use contract assertions to guide formal verification and correctness proofs for a given software component.

The above priority ordering is consistent with the one provided by Bjarne Stroustrup in [P1711R0].

Over the years, we have realized that trying to specify a Contracts feature that adequately addresses all five categories above in the initial version that we ship is not a viable strategy. Instead, we decided to ship an MVP (minimal viable product): an initial set of functionality and a platform upon which WG21 can agree and that can be extended later. By definition, an MVP cannot satisfy all known use cases but is designed to satisfy basic use cases from the start and simultaneously to allow satisfying other important use cases via future extensions to the C++ Standard.

> I am a great fan of the incremental approach — getting a minimal change in place and then improving it based on feedback. I consider that engineering as opposed to the ideal of getting a change perfect in advance — which I consider naive and at odds with reality.
>
> *Bjarne Stroustrup*[5]

This approach is a tried and tested recipe for successfully delivering powerful new language features to the C++ Standard. For example, C++11 introduced the initial version of `constexpr` with the restriction that a `constexpr` function could contain only a single `return` statement and nothing more. This strategy was limiting but useful to get implementation and usage experience with functions that can be evaluated during constant evaluation. C++14 added more features to `constexpr`, as did C++17 and C++20. Our goal is to follow the same approach with Contracts for C++.

We have thus decided to focus on the first three categories of needs — *documentation*, *runtime checking*, and *static analysis*, in that order of priority — in this initial version of Contracts for C++. However, with a feature well-specified to support these three categories, all the other categories of use cases are enabled as well and can be addressed via vendor extensions and/or future extensions to the C++ Standard.

Documentation is a crucial need: A plain-language contract is no good if the user of a library does not know that it exists and thus cannot follow it. To improve on the current situation where plain-language contracts are described in code comments is relatively low-hanging fruit for a Contracts facility: All that is required is a uniform standard syntax for contract assertions that can be placed on function definitions and parsed by humans, compilers, IDEs (enabling them to display hints about the contract to the developer as they type), and other tooling. [P2900R14] provides such a syntax (see Section 3.2). The significant value gained by providing just that is discussed in more detail in Section 3 of [P3376R0].

While well-documented code can greatly increase a user's ability to write code with fewer bugs and a standard syntax for such documentation that can be recognized by IDEs and other tooling greatly improves that documentation's visibility, extensive experience has shown the fairly low limit to the effectiveness of documentation at ensuring that users actually use a library correctly. *Runtime checking* provides a concrete mechanism to take contract assertions and make a program that, when run, can tell us if it is incorrect, resulting in identifying bugs as they happen so they can be fixed and incrementally improving a program toward complete correctness.

The feature set provided in [P2900R14] is, therefore, dominated by features that provide the required flexibility, scalability, and portability for runtime checking of contract assertions across all different kinds of C++ codebases: different *evaluation semantics* covering all important use cases (Section 3.5.4) that can be flexibly selected at compile time, link time, load time, or run time on a per-program, per-translation-unit or even per-assertion basis (Section 3.5.5) as well as a global, replaceable contract-violation handler that enables many different strategies to mitigate contract violations (Section 3.5.9).

Prioritizing the needs of *static analysis* tools when designing a Contracts feature for C++ was first proposed in [N4319], based on previous experience with Microsoft SAL and Code Contracts. While enabling correctness checks at run time is the predominant focus of the proposed Contracts feature, static analysis is another important tool to find program defects. In particular, static analysis helps

[5] Evolution Reflector, 2019-06-02

find such defects earlier in the development process and can identify edge cases that might be missed by tests and rarely seen in production. Static analysis is, therefore, complementary to runtime checking.

Static analysis tools are significantly aided by the Contracts feature proposed in [P2900R14], and this concept is discussed in more detail in [P3376R0] and [P3386R0]. Static-analysis tools are already successfully using assertion macros today for control flow analysis and range analysis among other things (see Section 3 of [P3386R0]). However, because assertion macros cannot be placed on declarations (i.e., the tool needs to inspect the definition, which might be unavailable or too complex to reason about) and are not visible to the tool when they are switched off and thus textually removed and due to the lack of a standard syntax (many projects use custom assertion macros instead of the standard `assert` macro), the usefulness of static-analysis tools is limited. Vendor-specific annotations can partially overcome these limitations but are not portable and have their own limitations. [P2900R14] instantly provides more value for such tools by removing all the above limitations.

The *optimization* use case can provide significant benefit in the form of increased performance, but when used incorrectly, this approach can too easily lead to additional undefined behavior in a C++ program. Prioritizing optimization is, therefore, at odds with the primary goal of the proposed Contracts feature to *help identify program defects* and the general push toward more *safety* in C++ (see Section 2.7). More discussion of this use case can be found below: Section 2.3 provides our motivation for excluding *assumptions* from [P2900R14], and Section 3.5.4 contains a record of how this decision came to be. Both sections contain references to the many relevant papers.

The *verification* use case has been a much-discussed topic during the development of [P2900R14]. [P3362R0] makes the case that using contract assertions for verification and correctness proofs is a desirable direction but requires the contract predicate to be restricted to expressions that can be proven to have no side effects and to avoid certain forms of undefined behavior, such as the *strict contracts* proposed in [P2680R1] and again in [P3285R0]. However, we found that strict contracts seem to be too restrictive, regarding which predicates they allow, to be practically useful for the more important *runtime checking* use case (see Section 4 of [P3386R0]). We have thus decided not to pursue this direction. A more detailed discussion of this tradeoff and a record of how this decision came to be can be found in Section 3.6.1.

2.2 Proposed Features

Although intended as an MVP, [P2900R14] does not necessarily represent the smallest possible feature set that could be meaningfully standardized. For example, [P1607R1] proposed a much smaller feature set, and [P2900R14] is commonly criticized as being too large. However, [P2900R14] is indeed *minimal* in the sense that it represents the smallest possible feature set that, as a whole, could achieve SG21 consensus within the existing WG21 standardization process.

Several features have been added to the Contracts proposal between its first, smaller version [P2388R0] and the current version [P2900R14], notably user-defined contract-violation handlers, the *observe* and *quick-enforce* semantics, support for `pre` and `post` on coroutines, and implicit `const`-ness of entities declared outside a contract predicate (informally called "`const`-ification"). Each of these additions was necessary to overcome limitations that, at least for some members of SG21, would have made the entire proposal unviable and therefore unable to achieve the necessary consensus to

proceed toward standardization. In this section, we provide a brief motivation for the scope of the feature set as it is in [P2900R14] today.

[P2900R14] proposes three kinds of contract assertions: *precondition assertions*, *postcondition assertions*, and *assertion statements*.

The purpose of precondition assertions is to identify contract violations when *calling* a function: For the caller, they can act as verification that they called the function correctly, and for the callee, they can establish conditions on which the implementation of the function can rely. Precondition assertions are a key feature of [P2900R14] because they provide the syntax necessary to express postconditions on function *declarations* rather than being limited to function *definitions*, which, as we have seen above, is an essential part of the value that [P2900R14] provides over existing assertion facilities.

The purpose of postcondition assertions is a mirror image of the above: For the caller, they can establish conditions that the code following the function call can rely on, and for the callee, they can act as verification that the function has been implemented correctly. However, postcondition assertions have a less favorable cost-to-benefit ratio than postcondition assertions. They are significantly harder to implement (see [P3460R0]), while at the same time we expect them to be less widely used; in fact, much of their utility can also be obtained with a good unit test coverage. SG21, therefore, considered shipping a Contracts facility without postcondition assertions.

SG21, Teleconference, 2021-12-14, Poll

Postconditions should be in the MVP at this time.

SF	F	N	A	SA
1	7	3	4	1

Result: Marginal consensus (if at all)

Despite the marginal consensus in the poll above, the Contracts proposal proceeded *with* postcondition assertions and later gained strong SG21 consensus; even those members generally not in favor of postconditions believe at this point that consensus will not be achieved on a feature without them. After several issues with odr-using nonreference parameters in postcondition assertions came up ([P3484R1], [P3487R0], [P3489R0]), SG21 once again considered removing postcondition assertions from the Contracts proposal, but had strong consensus not to do so (for context of this poll, see Section 3.4.5) and instead to approve targeted fixes for the known issues.

SG21, Teleconference, 2024-11-07, Poll 1

For P2900, remove postconditions, as proposed in P3487R0 Option R1.

SF	F	N	A	SA
0	0	3	9	7

Result: Consensus against

Assertion statements are a straightforward and powerful tool to introduce correctness checks at arbitrary points during the evaluation of a program, either wherever a statement can be placed or, when embedded within an immediately invoked lambda, as parts of an arbitrary expression. By

using assertion statements, preconditions and postconditions of third-party functions that are not annotated with function contract assertions can be implemented. Similarly, the state of a program can be verified using assertion statements surrounding the suspension points of a coroutine. Overall, they provide a foundation to introduce a correctness check in places not facilitated by the rest of the Contracts feature.

The proposed set of *evaluation semantics* (see Section 3.5.4) is necessary to cover all known important use cases for runtime checking of contract assertions. A contract-violation handler (see Section 3.5.9) is necessary to provide a way to mitigate contract violations at run time.

2.3 Features Not Proposed

The ability to specify precondition and postcondition assertions on virtual functions is important for adopting contract assertions into codebases that make use of runtime polymorphism. Nevertheless, [P2900R14] intentionally omits this feature, and the reasons for and history of this omission can be found below in Section 3.3.2. We expect that this feature will be added as a future extension to the Contracts proposal.

Similarly, [P2900R14] does not provide the ability to specify precondition and postcondition assertions on pointers to functions, pointers to member functions, or type aliases for such types despite that functionality having been requested repeatedly. The motivation for this decision is provided below in Section 3.3.6 and in much more detail in [P3327R0].

Further, [P2900R14] does not offer the ability to refer to the original, "old" values of parameters and other entities at the time a function was called during the evaluation of that function's postcondition assertions. This limitation makes certain common postconditions inexpressible with [P2900R14], e.g., the postcondition of `std::vector::push_back` that if the function returns normally, the size of the vector will be incremented by one. Nevertheless, this functionality was never considered essential for an MVP and is instead planned as a future extension. A possible specification for this extension is the proposal for postcondition captures [P3098R0].

Another future extension that will significantly expand the functionality of the proposed Contracts feature but is not essential to have in its first version is the ability to place *labels* or other forms of additional annotation directly on contract assertions. Such labels could, for example, express the desired evaluation semantic directly on the contract assertion, assign an assertion level to it, or specify other properties of a contract assertion and how they map to an evaluation semantic, which can be made configurable by the user. A first exploration of contract labels can be found in [P2755R1]; a fleshed-out proposal that is currently under consideration by SG21 can be found in [P3400R0].

The absence of support for class invariants in particular has caused mild disappointment over the scope of the Contracts proposal, in particular because other programming languages that have a Contracts facility as a core-language feature such as Eiffel, D, and Ada (see Section 2.5), do offer class invariant assertions. As pointed out in [P2755R1], offering this functionality is highly desirable, but the wide range of open questions regarding how exactly this would work in C++ will need time and effort to resolve. Holding up the entire Contracts feature until class invariants for C++ are successfully specified does not seem like the best tradeoff for the C++ community and is not in the spirit of providing an MVP.

Procedural function interfaces, first proposed in [P0465R0], extend precondition and postcondition assertions by offering the ability to express contracts that are inexpressible with a boolean predicate and contracts that specify forms of exiting a function other than by returning normally, in particular, contracts on functions exiting via an exception. (Note that in [P2900R14], postcondition assertions are checked *only* when a function returns normally, not when it exits via an exception.)

The last noteworthy omission from [P2900R14] that we explicitly discuss in this paper is that of an *assume* semantic, which is an unchecked evaluation semantic that is distinct from *ignore* and in which the truth of the predicate is assumed, allowing the compiler to optimize the program under this assumption; if the predicate is not `true` when evaluated, the behavior is undefined. The *assume* semantic is required to address the *optimization* use case (see Section 2.1 above) as part of a Contracts facility.

We know that assumptions, when used carefully and correctly, can lead to measurable performance gains (see [P2646R0]). These gains make assumptions a valuable feature, which is why the feature was added to C++ independently from Contracts via the `[[assume]]` attribute ([P1774R8]). We believe that providing an *assume* semantic as part of a Contracts facility would make assumptions more customizable, scalable, and usable on interfaces than when using `[[assume]]`. More importantly, by having the ability to easily take assumptions and instead observe or enforce them, we provide a mechanism to leverage assumptions with vastly increased chances of correctness (see [P2064R0]).

However, we do not include assumptions in [P2900R14] for three reasons. The first is that we consider the *optimization* use case less of a priority than the *runtime checking* use case. The second is that the `[[assume]]` attribute is already part of the Standard, and, therefore, a temporary workaround is available while *assume* is not yet a Standard contract-evaluation semantic. Finally, the third reason is that the ability of *assume* to add undefined behavior to an existing program is at odds with the primary goal of the proposed Contracts feature to help identify program defects and to push toward more safety in C++ (see Section 2.7); thus, the inclusion of *assume* could make the entire proposal unnecessarily controversial. Relegating the *assume* semantic to a possible future extension is, therefore, a conservative approach.

While *assume* is not part of [P2900R14], a compiler vendor is free to provide such a semantic as an extension since [P2900R14] explicitly allows such vendor-specific evaluation semantics.

The history of the *assume* semantic so far, and all relevant decisions in EWG and SG21 related to this feature, are documented as part of the history of evaluation semantics in Section 3.5.4.

2.4 How to Use Contracts

Even without any contract assertions, every function written in C++ has a *contract* that defines how that function is expected to interoperate with other components in a correct program. This contract might be

- documented in comments in the source code
- documented in a separate document, for example, the C++ Standard, which contains extensive specifications of the preconditions and postconditions of all functions in the C++ Standard Library

- part of a coding convention or guideline, for example, that a `const` member function should be thread-safe
- entirely implicit, being the product of the parameters a function takes and the code that constitutes its implementation

and anywhere in between. Each such *plain-language contract* necessarily contains two key aspects. First is the (possibly empty) set of preconditions, i.e., the requirements that the caller must satisfy when invoking the function. Preconditions include constraints on the input arguments as well as on the object and program state. Second is the essential behavior, i.e., what the function promises to do given valid input. Essential behavior includes postconditions — return values, changes in program or object state, thrown exceptions, and so on — and behavioral guarantees, such as algorithmic complexity or thread safety.

Breach of contract necessarily indicates a defect in the program — i.e., a defect in the calling code when preconditions are not met and a defect in the implementation if the function doesn't adhere to its essential behavior. While a programmer might employ many tools to ensure that their function conforms to its specification (i.e., fulfills its plain-language contract), from unit and integration testing to formal verification, preventing or even detecting client misuse of a library function is more difficult. We might then be asked, "Why have any preconditions at all?"

Function preconditions serve several important purposes (see [P1743R0]).

- **Feasibility**: Defining behavior for all inputs might be impossible. For example, trying to define the behavior of `std::sort` for comparators that do not impose a strict weak ordering on the elements of the range would require checking that condition, which is not possible in the general case.
- **Efficiency**: Defining behavior for all inputs might impose an unacceptable performance penalty. For example, should `std::lower_bound` need to check that the range is partitioned with respect to the value being searched, its asymptotic complexity would necessarily degrade from $O(logN)$ to $O(N)$. Furthermore, this penalty would be present on every call even in cases where the caller ensures that property by sorting the range prior to performing lookups.
- **Reliability**: Handling nonsense inputs might mask program defects. If `std::vector::pop_back` were just a no-op when the vector is empty, a latent defect in the caller that leads to calling `pop_back` more times than there are elements in the `vector` might lie dormant for a long time and manifest in a completely different part of the program, possibly with dire consequences.
- **Maintainability**: Handling all possible inputs requires a lot of code dedicated to detecting which inputs are acceptable and which are abnormal. Handling abnormal inputs also requires all clients to handle far more error paths.
- **Extensibility**: Once the behavior of a function is defined for obviously incorrect inputs, changing that behavior without breaking backward compatibility is no longer possible.

Designing functions with appropriate preconditions has a positive effect on the overall quality of software. A good library function contract is minimal but complete. It should handle all reasonable inputs while refraining from defining specific behaviors for inputs that do not make sense, leaving that behavior undefined.

A good example of this approach is `std::string_view::remove_prefix(size_type count)`, which defines the behavior for all values of `count` in the closed interval of `[0 .. size()]`. Handling `0` and `size()` often allows the calling code to be simpler than it would have been if these edge cases weren't handled and keeps the implementation of `remove_prefix` the same. At the same time, calling `remove_prefix` with `count > size()` is a contract violation since such a value for `count` is highly unusual and likely indicates a defect in the code surrounding a call to `remove_prefix`.

In some circumstances, library clients are well served if the library provides dual APIs with similar functionality — one function with preconditions (i.e., a narrow contract) and one without preconditions (i.e., a wide contract) — but reports an error for inputs that would lead to a contract violation. For example, a networking library might provide such a dual API for adding a field to an `HttpHeader`, alongside a means to check whether a name-value pair would be a valid header field:

```
class HttpHeader {
    // [...]

    // Return true if name and value form a valid header field and
    // false otherwise.
    static bool isValidField(std::string_view name, std::string_view value);

    // Add a header field with the specified name and value.  The behavior
    // is undefined unless isValidField(name, value).
    void addField(std::string_view name, std::string_view value);

    // Add a header field with the specified name and value and return
    // true if isValidField(name, value), and return false otherwise.
    bool addFieldIfValid(std::string_view name, std::string_view value);

    // [...]
};
```

Having such an API serves a wide variety of clients. Those that know that their field name and value are valid can invoke `addField` directly and avoid the performance penalty of rechecking what they already know is true, those that need to validate their field prior to deciding whether they want to add that field or not based on other information can use `isValidField` followed by `addField`, and those that want to validate and add the field in one go can use `addFieldIfValid` to do both in a single function call.

Irrespective of the API choices, for as long as humans develop the software, most functions will eventually be misused and called out of contract. Contract assertions allow detecting such contract violations early in the development cycle, allowing a reduction in the defect incidence in production software. In the example of `HttpHeader`, the library developer would add a precondition annotation to the `addField` function:

```
class HttpHeader {
    // [...]

    // Add a header field with the specified name and value.  The behavior
    // is undefined unless isValidField(name, value).
```

```
    void addField(std::string_view name, std::string_view value)
        pre(isValidField(name, value));
};
```

If the library client were to write code that would feed untrusted external input to a call to `addField` and to configure their compiler such that the contract semantic for the precondition assertion is anything but `ignore`, they would be notified of the precondition violation and fix their bug. Note that the distinction between what can be considered trusted versus untrusted data is highly application- and domain-specific. [P2053R1] takes a deep dive into that topic.

An important aspect of contract assertions — from which both their greatest strengths and some potential for misuse come — is that they are, in essence, redundant. In a defect-free program, all the contract assertions pass, so the checks that they make have no effect on the behavior of such a program, meaning that compiling the same program with the contract assertions having only the *ignore* semantic would lead to attaining a performance boost. If the program misuses contract assertions, however, the behavior of the program will change, and it will no longer adhere to its specification. Such misuses generally stem from writing contract assertions that affect the program's essential functionality. For example, a program containing a contract assertion with a predicate having a meaningful effect on the core functionality — or in other words, one that contains a destructive side effect as opposed to a benign one (see [P2053R0] Appendix A) — might produce a completely incorrect result. However, if programmers follow just a few best practices for writing contract assertions (i.e., avoiding contract predicates with destructive side effects, as defined in Section 3.1.2 of [P2900R14], and not using them for control flow or input validation), using functions with narrow contracts where appropriate coupled with contract assertions for detecting client misuse will allow them to build faster, easier-to-understand, and more reliable software.

Once we have a design, the question then becomes how to encode contract checks into our program with the tools provided by [P2900R14]. For every narrow precondition in the plain-language contract of a function, which can include someone saying "Just don't use it that way," we should assess a few key properties to identify whether to encode it using a precondition assertion.

- Do we have a relatively simple condition that is easily checked? If so, add a precondition assertion for this condition:

  ```
  void vector::operator[] (size_t index)
    pre (index < size();
  ```

- Do we have a more involved check that requires data that is not accessible? If so, we can consider widening lower-level APIs to make that data available.

- Is the check going to make our program incorrect because it violates the performance requirements of our program? In particular, do we have complexity requirements, for example, the precondition of `binary_search`, which is $O(N)$, that the input range is sorted, which is $O(\log N)$? If so, we need to wait for the addition of labels (as described in [P3400R0]) in a future version of the C++ Standard to annotate the check as one that should be run only in builds where such costs are managed. [P2900R14] does not yet support this use case.

- Is the only algorithm to check the condition one that would be destructive? If so, we need labels to call out checks that should not be evaluated at run time. [P2900R14] does not yet support this use case.

A similar operation can be performed for postconditions. The utility of *deploying* postcondition assertions at run time diminishes with better unit testing, which, of course, we should be doing as well. However, postcondition assertions are still useful since unit testing covers only function calls with a particular set of inputs, whereas postcondition checks will apply to every call of the function performed anywhere in the program when enabled.

Note that adding these contract assertions does not, on its own, require that any changes happen in any of the programs we actually run. For each environment in which our code might run, we will configure our compiler in different ways to maximize the benefit of contract assertions and manage our costs and risks appropriately.

- In our development systems, we'll run code with all our checks *enforced* except when performance testing. This setting will catch bugs quickly, long before they become a production problem. We can then fix our bugs, redeploy, and enjoy.
- As we prepare to release to production, we will do our performance testing with contracts having the *enforce* or *ignore* semantic. For very large-scale projects, we'll consider the *quick-enforce* semantic as well. If the performance degradation of checking our contracts is small, then we'll consider just deploying that to production; it is the safer choice. If not, we'll deploy the build with contract assertions ignored, but the pressure will increase tremendously to enforce those contracts in all our environments before production.
- Our unit tests should be run twice, once with as many checks *enforced* as possible and once with the same configuration as our production environment. (This second build need not be run with the same frequency.) The two builds together will help weed out any destructive contract assertions we might have inadvertently written, while the maximally checked build will greatly increase the scope of problems our unit tests are capable of finding.
- When we later decide to take a running system and try to add even more contract assertions to it, we will have a use for the *observe* semantic. We can refer to [P2755R1] and [P3400R0] for several use cases where this semantic can be applied as well as the kinds of labels that will be needed to fully enable that aspect of this feature.

Across the projects we maintain, write, or use from a library, a contract-violation handler will report error information usefully back to us so that bugs can be promptly addressed.

Plain-language contracts have many other uses and ways that they can be leveraged them that we have and will continue to explore, all of which will build on the foundation provided by [P2900R14]. The above is a start, but effective use of Contracts will breed further effective use of Contracts.

Importantly, we must always remember that it just takes one contract assertion to begin finding bugs (or to identify that a program is already robust in some particular way).

2.5 Prior Art

The concept of Design by Contract was coined by Bertrand Meyer when he conceived and developed the Eiffel programming language. Eiffel became available in 1986 and had support for preconditions, postconditions, and class invariants.

Other programming languages that include a Contracts facility as a core-language feature include D, Dafny, and Ada. The D programming language was first released in 2001 with support for preconditions, postconditions, and class invariants included from the first version. Dafny was first released in 2009 as an educational language and includes support for a wide array of contract specifications, including preconditions, postconditions, loop invariants, loop variants, termination specifications, and read/write framing specifications. In 2012, preconditions, postconditions, class invariants, and loop invariants were added to the Ada programming language.

Note that the Contracts facilities in these programming languages were designed with different use cases in mind, reflecting different possible answers to the question of the purpose of a Contracts facility (see Section 2.1). For example, in D, contracts are intended primarily for dynamic verification (i.e., runtime checking); in Ada, contracts are leveraged by both static and dynamic verification; and in Dafny, contracts are used for formal correctness proofs based on the general framework of Hoare logic.

While relatively few languages — none of the top twenty on the TIOBE index — have a core-language Contracts facility, many more programming languages have some form of support for contract checking as a library feature. Noteworthy examples are Microsoft Code Contracts, which is a framework for specifying preconditions, postconditions, and object invariants for C# and other .NET Framework code (discontinued in .NET 5), and Java, which has several actively maintained annotation libraries providing similar functionality.

As a more primitive form of Contracts facility, assertions have been available for a long time. The `assert` macro has been available in C since ANSI C (1989) and in C++ from its inception. Nearly every programming language provides an assertion in the form of `assert(`*`expression`*`)` or similar.

The exact semantics of such assertions vary from language to language, as every language has its own needs, paradigms, and acceptable mitigation strategies. For example, in C and C++, `assert` is a macro that can be enabled or disabled and the chosen mitigation strategy on failure is program termination. On the other hand, in languages like Python or Java, the appropriate mitigation strategy is instead to throw an `AssertionError` exception, since these languages do not have to guard against undefined behavior and are generally written such that all code is exception-aware. The mitigation strategy for contract violations (i.e., program defects) in those languages is, therefore, identical to that for external errors, such as invalid input, which is a noteworthy difference to Contracts in C++ (see [P2900R14], Design Principle 12, Contract Assertions Are Not Flow Control).

Also note that, while assertions can usually be switched on or off in most programming languages, the granularity with which this switching is possible also varies from language to language and reflects language-specific characteristics. In C and C++, which have header files, translation units, and a linker, assertions are enabled or disabled per translation unit[6]; in languages like Python or

[6]Note that not changing the behavior of `assert()` is just general practice since we can re-`#include` the `assert.h` or `<cassert>` header with a different state for the `NDEBUG` macro and enable or disable assertions within a single translation unit.

Java, assertions are enabled or disabled globally; in Eiffel, enabling or disabling contract checks is possible on a per-class level.

Another type of feature with functionality adjacent to the Contracts facilities discussed so far are annotations — such as "this pointer will not be null" or "this array will not be empty" — that can be used to express contracts on function parameters, classes, and other components. This feature typically takes the form of specific annotations that spell out the expected or ensured property rather than generic assertions that contain a boolean property. Examples of such annotation facilities are Microsoft's SAL (Source-code Annotation Language) and the bounds-safety extension in Clang. Again, such facilities can have a different focus and be designed for static or dynamic verification or both.

2.6 A Short History of Contracts for C++

The history of attempts to standardize a Contracts feature for C++ spans over two decades. Within this history, we can distinguish four distinct phases.

Phase One: D-Like Contracts. The first formal proposal to add a Contracts feature to C++, [N1613], appeared in 2004 and was heavily influenced by the D programming language. Preconditions and postconditions could be specified as statements inside `in` and `out` blocks, respectively:

```
int foo (int& i)
in {
  // precondition block
  i > 0;
}
out {
  // postcondition block
  i == in i + 1;
  return % 2 == 0;
}
do {
  // normal function body
  ++i;
  return 4;
}
```

The proposal also provided syntax to specify class invariants and a way to specify inline the desired fashion in which contract violations should be handled at run time:

```
class vector {
  invariant {  // class invariants
    size() <= max_size();
    distance(begin(), end()) == size();
    // ...
  }
}

double sqrt( double r )
in {
  r > 0.0: throw bad_input();  // runtime handler
}
```

This proposal went through four subsequent revisions ([N1669], [N1773], [N1866], [N1962]). The keywords `in` and `out` were replaced by `precondition` and `postcondition`, respectively; the way to refer to the "old" value of a variable in a postcondition (the value at the time when the function was called) was changed first to `std::old(x)`, then `__old(x)`, and finally `oldof(x)`; the concept of "importance ordering" (an early version of checking levels) was added and then removed again; and various smaller changes to the design were undertaken. This design was implemented by the Digital Mars compiler,[7] using the keywords `__in`, `__out`, `__body`, and `__invariant` instead of `in`, `out`, `do`, and `invariant`, respectively. Ultimately, however, this design direction was not pursued further.

Phase Two: Macro Contracts. The next formal proposal for a Contracts feature for C++, [N3604], appeared in 2013. It was based on Bloomberg's BSL library and followed a different design. The ability to express preconditions, postconditions, and invariants on interfaces was abandoned in favor of assertion macros that could be placed inside a function body:

```
std::size_t strlen(const char* str) {
  pre_assert(str);  // precondition check
  // ...return string length
}
```

Unlike the existing `assert` macro, [N3604] proposed a set of four macros with three checking levels ("safe," "debug," and "optimized," designated by a suffix) and a default level, different build modes that enable or disable different checks depending on their checking levels, and a single, user-defined contract-violation handler.

This proposal went through several subsequent revisions ([N3753], [N3818], [N3877], [N3963], [N3997], [N4075], [N4135], [N4253], [N4378]), during which numerous aspects of the proposal were changed, such as the names of the macros (from `pre_assert(x)` to `CONTRACT_ASSERT(x)` to finally `contract_assert(x)`, which interestingly is the same syntax used by [P2900R14] for an assertion inside a function body) and the API for the contract-violation handler. Ultimately, the proposal failed to get consensus in WG21 plenary.

Phase Three: C++2a Contracts. At the time when the Macro Contracts proposal failed in plenary, several other parties were working on alternative designs for a Contracts feature for C++. Explorations of the design space were undertaken in [N4110], [N4319], [P0147R0], [P0166R0], and [P0247R0]. Early Phase Three Contracts proposals were [N4248], [N4293], [N4415], [N4435], and [P0287R0].

The different parties involved in these proposals and in the failed Macro Contracts proposal eventually joined forces and produced a merged proposal [P0246R0] (which became known as the Contracts Merged Proposal) and another merged proposal involving even more parties [P0380R1], which ultimately led to the so-called *C++2a Contracts* proposal [P0542R5], which achieved consensus in WG21 Plenary and was added to the C++20 Working Paper in 2018. Two improvements on top of [P0542R5] — [P1289R1] and [P1323R2] — were also added to the C++20 Working Paper.

The C++2a Contracts proposal offered precondition and postcondition assertions that can be added to function declarations and provided assertion statements inside function definitions, thus

[7] See https://www.digitalmars.com/ctg/contract.html.

supplying the same three kinds of contract assertions proposed by [P2900R14]. The syntax was an attribute-like (but not actually attribute) syntax:

```
void f(int x, int y)
  [[expects: x > 0]]          // precondition
  [[expects: y != 0]]         // precondition
  [[ensures r: r > x + y]]    // postcondition
{
  int n = get_n();
  [[assert: n > 0]];          // assertion
  return x + y + n;
}
```

This proposal was implemented by the GCC compiler (see [P1680R0]).

While many elements of C++2a Contracts — such as the three kinds of contract assertions and the contract-violation handler — are still present in [P2900R14] in some form, other elements of the design were markedly different; for example, C++2a Contracts contained *contract levels* and *build modes* to choose the evaluation semantic of a contract assertion, features absent in [P2900R14].

As the technical work on C++20 was wrapping up, concerns were raised over the C++2a Contracts specification in the C++20 Working Paper. Controversy arose over various aspects of the specification, such as the proposed build modes, the meaning of `axiom`, and an unchecked contract assertion being *assumed* by default (meaning that the behavior is undefined if its predicate would evaluate to `false`). After a series of heated discussions at the June 2019 WG21 meeting in Cologne (which marked the feature-freeze date for C++20), a significant design change ([P1607R1]) was adopted by EWG at the last minute. This late and massive change eroded trust in the proposed design and ultimately caused WG21 consensus on the entire feature to evaporate. [P1823R0] proposed to remove the entire feature from the C++20 Working Paper and was approved by EWG.

EWG, Cologne, 2019-07-17, Poll

Withdraw P1607 and accept D1823R0 'Remove Contracts from C++20'

SF	F	N	A	SA
24	25	3	6	5

Result: Consensus

At the end of the Cologne meeting, [P1823R0] was then approved by WG21 Plenary, sealing the fate for C++2a Contracts.

Phase Four: The Contracts MVP. After C++2a Contracts was removed from the C++20 Working Paper, SG21 (the Contracts Study Group) was set up to start over and produce a new Contracts proposal that can get consensus in WG21. [P2900R14] is the product of this work, which was conducted in SG21 over the last five years.

Phase Four is the focus of this paper. We provide here the complete history of developing [P2900R14] in SG21, including all papers considered and all polls taken. When necessary to understand of how a particular decision came about, history from the previous phases is given but generally only as a brief summary with relevant paper references and without detailed poll results.

SG21's work started with identifying why C++2a Contracts had failed. [P2076R0] found that the main reasons were disagreements on the following topics: continuation, build levels, assumption, and the use of global toggles and literal semantics to select the desired behavior. Given those past disagreements and the inability to resolve them effectively, SG21 decided to start designing a new Contracts facility from scratch — informed, of course, by the previous proposals but not tied to any particular design decisions in those proposals.

Before starting any design work, SG21 spent a considerable amount of effort on identifying and prioritizing use cases that a new Contracts MVP should address ([P1995R1], [P2114R0], [P2185R0]) and establishing the required nomenclature ([P2038R0]). A minimum viable feature set was then proposed in [P2182R1] and was well received by SG21.

SG21, Teleconference, 2021-01-19, Poll

We are confident that the paper represents a reasonable specification of a valid Programming Model, and the authors can proceed (modulo a discussion of how to optimize if possible).

SF	F	N	A	SA
4	9	3	0	0

Result: Consensus

The first formal proposal in this phase was [P2388R0]; it was likewise well received by SG21.

SG21, Teleconference, 2021-07-20, Poll

SG21 would like to continue work on P2388R0 as the contract support for C++23 and proceed to develop wording.

SF	F	N	A	SA
8	4	2	0	0

Result: Consensus

P2388 went through multiple iterations. The last revision of that paper was [P2388R4]; consensus on the different components of its design were tracked separately in a "record of consensus" paper, the latest revision of which was [P2521R5].

In October 2021, SG21 decided to pursue more design work for the Contracts feature, even though those changes would cause Contracts to miss C++23.

SG21, Teleconference, 2021-10-12, Poll

We should pursue the design space of syntax for contracts, semantics of predicates, semantics of postconditions, etc. before approaching EWG, which would preclude shipping in C++23.
Result: No objection to unanimous consent

At the November 2022 meeting in Kona, SG21 decided to target C++26 with the new Contracts proposal and adopted a roadmap, [P2695R1], designed to stay focused on achieving this goal.

SG21, Kona, 2022-11-12, Poll

Adopt D2695R0 as presented as the roadmap for working on a Contracts MVP targeting the C++26 IS, which will need consensus in SG21 to revise. Consensus on technical decisions takes precedence over the roadmap; we will amend the roadmap accordingly if a technical decision requires it.

SF	F	N	A	SA
10	10	5	1	1

Result: Consensus

Going forward, the status quo of the Contracts proposal was tracked in a new Contracts paper with the paper number P2900, and the more detailed rationale and record of consensus is tracked in *this* paper, P2899.

[P2659R2] proposed to publish the last version of C++2a Contracts approved by EWG (the last version from the C++20 Working Paper, with [P1607R1] and [P1344R1] additionally applied) as a TS to gain early experience with the feature while we continue to work on a new Contracts proposal. However, publishing such a TS was seen as a distraction from SG21 work and was rejected by SG21.

SG21, Teleconference, 2022-12-01, Poll

Given that a Contracts TS as proposed in D2659R2 cannot be forwarded by SG21 to EWG without review, we choose to not pursue a Contracts TS at this time, in favour of pursuing the Contracts MVP as agreed on in the current SG21 roadmap (P2695R0).

SF	F	N	A	SA
14	1	3	0	0

Result: Consensus

The following year saw significant progress toward the technical work on the Contracts proposal, documented in later sections of this paper. Finally, Revision [P2900R6] of the Contracts proposal gained consensus in SG21 and was forwarded to EWG and LEWG for design review.

SG21, Teleconference, 2024-02-29, Poll

Forward P2900R6 to EWG and LEWG for design review with C++26 as the target ship vehicle.

SF	F	N	A	SA
13	4	1	0	1

Result: Consensus

The first round of design review in EWG and LEWG took place in March 2024 at the WG21 meeting in Tokyo and revealed that some open questions remained. LEWG did not take polls, but several LEWG members gave informal feedback, particularly around naming (which has since been addressed; see Section 3.7). EWG captured their open questions in ten polls. SG21 responded to all ten polls (see [P3197R0]) and addressed the issues raised via these polls in revision [P2900R10].

In [P3265R3], a member of SG21 and EWG suggested that, instead of targeting the IS, the proposal should target a TS. As motivation, the paper cites lack of implementation, deployment, and usage experience (which have since been addressed; see also Section 2.9), lack of WG21-wide consensus on certain design questions (many of which have since been addressed as well), and the ability of a contract predicate to exhibit undefined behavior when evaluated, just like any other C++ expression (see Section 3.6.1). Multiple response papers ([P3269R0], [P3276R0], [P3297R0]) have been published arguing that the design is correct as is and that delaying Contracts and shipping it as a TS would be a mistake; the Direction Group has also opined on this question ([P4000R0]). Even though EWG owns the question of ship vehicle for any new C++ language feature, SG21 was asked to provide a recommendation on this question, and consensus was against shipping Contracts as a TS.

SG21, Teleconference, 2024-05-30, Poll

SG21 recommends that the ship vehicle for the Contracts MVP be a TS and not the C++ IS (C++26 if possible or C++29 if C++26 is not possible)

SF	F	N	A	SA
1	3	3	12	10

Result: Consensus against

The second round of design review in EWG took place in June 2024 at the WG21 meeting in St. Louis. During this meeting, many outstanding design decisions have been solved, such as gaining EWG consensus for the ability of a contract-violation handler to throw an exception.

The final round of design review in EWG and LEWG took place in November 2024 at the WG21 meeting in Wrocław. At that meeting, both groups approved the design proposed in the revision reviewed, [P2900R11], and forwarded the proposal to the respective wording groups with C++26 as the target ship vehicle. Neither group requested any design changes at that stage.

EWG, Wrocław, 2024-11-18, Poll

P2900r11: send to CWG and LEWG for inclusion in C++26.

SF	F	N	A	SA
25	17	0	3	12

Result: Consensus

LEWG requested only the addition of a separate feature test macro for the library API (see Section 3.8).

LEWG, Wrocław, 2024-11-21, Poll

Add the library feature test macro "__cpp_lib_contracts" and forward P2900R11 to LWG for C++26.

SF	F	N	A	SA
23	9	1	0	5

Result: Consensus

Following design approval by EWG and LEWG, the Contracts proposal went into wording review by CWG and LWG. An initial round of CWG wording review in Wrocław uncovered a handful of minor design questions. These questions were resolved in [P3520R0]; one more oversight was resolved in [P3510R2]. All those papers were design-approved by SG21 and EWG at the end of the Wrocław meeting and incorporated into revision [P2900R12].

In the lead-up to the February 2025 WG21 meeting in Hagenberg, a number of concerns were raised over the Contracts proposal. These concerns are listed in [P3506R0] and [P3573R0]; most of them were not new issues but restatements of known, sustained opposition to the Contracts design. A comprehensive response to all these concerns was published in [P3591R0].

Another source of sustained opposition that was repeatedly voiced in WG21 but not captured in any published paper was the claim that the proposed Contracts facility was not "safe" because contract checks could be *observed* or even *ignored*, thus allowing control flow to progress into "unsafe" code. To address this concern, [P3500R0] and [P3578R0] were published. These papers explained the crucial difference between *functional safety* and *language safety* and the related role of contract assertions (see also Section 2.7 of this paper). [P3500R0] further presented various known workarounds to achieve "always enforced" contract assertions within the current design; [P3400R0] proposed a future extension to [P2900R14] that enables this use case via *labels*.

All these issues were discussed in detail at the EWG session in Hagenberg. A summary of the discussion and relevant poll results on the individual issues can be found in their respective subsections within Section 3 of this paper. At the end of this session, EWG took a poll on whether, given the sustained opposition, the Contracts proposal should be withdrawn from consideration for C++26, and consensus was not to do so.

EWG, Hagenberg, 2025-02-11, Poll

P2900: remove P2900 from CWG's consideration for C++26, find a different ship vehicle.

SF	F	N	A	SA
9	8	3	19	41

Result: Consensus against

Following this decision, the wording of the proposal was approved at the Hagenberg meeting by CWG on 2025-02-12 (no poll taken) and by LWG on 2025-02-13.

LWG, Hagenberg, 2025-02-13, Poll

Put P2900R14 into C++26 (to be moved by core).

F	N	A
13	0	0

Result: Consensus

At the plenary session of the Hagenberg meeting on 2025-02-16, [P2900R14] was adopted into the C++26 Working Paper with strong consensus, thus successfully concluding the many years of work that went into this proposal.

2.7 Contracts and Safety

Improving the safety and security of C++ is a major challenge for the evolution of the language ([Bastien2023]). Governments and regulatory bodies have called for the tech industry to move away from C and C++ and toward languages with better memory safety ([NSA2022], [CR2023], [CISA2023]), Rust being a frequently cited example. The lack of safety in C++ is considered by some to be an existential threat to C++ as a programming language for new projects; more recently, a clear trend has emerged in which large tech companies, such as Microsoft ([Claburn2023]) and Google ([Lakshmanan2024]), are moving away from C++ and toward Rust and other languages with better memory safety.

Simultaneously, evolving the C++ Standard toward more safety is an ongoing effort in WG21. For C++23, a common cause of undefined behavior in range-based `for` loops has been removed ([P2012R2]). For C++26, executing a trivial infinite loop ([P2809R3]) and, in some cases,[8] reading an uninitialized value ([P2795R5]) will no longer be undefined behavior. Efforts to tame undefined behavior at a larger scale include the current work on safety profiles ([P3274R0]) and adding a Rust-like borrow checker to C++ ([P3390R0]). Outside the C++ Standards Committee, compiler vendors are shipping solutions in this space — e.g., the hardening modes in libc++[9] and the bounds safety extension in Clang.[10]

The Contracts feature proposed in [P2900R14] is aimed squarely at making C++ safer and intends to provide the foundation for resolving the problem. Nevertheless, two questions are commonly asked.

1. Does the Contracts feature address safety in C++?
2. With so many other safety features competing for WG21's focus, why should the Contracts feature take priority?

The first question has caused some controversy. Proponents of the proposed Contracts feature point out that it provides a significant leap forward for safety ([P3297R1]), yet others have claimed that the feature does not help with safety at all and even that discussing Contracts in the context of safety is a category error. This misunderstanding rests on different people meaning different things when they discuss *safety*.

[8] I.e., if the variable is default-initialized, of scalar type, and has automatic storage duration

[9] See https://libcxx.llvm.org/Hardening.html and [P3471R4].

[10] See https://clang.llvm.org/docs/BoundsSafety.html.

According to the definition in [Carruth2023], *safety* is characterized by invariants or limits on program behavior in the face of bugs; safety bugs occur when some aspect of program behavior has no invariants or limits. According to the definition in [Abrahams2023], a safe operation is one that cannot cause undefined behavior; a safe language has only safe operations. All the efforts cited above to make C++ more safe are talking either about this general definition of safety as *language safety* or about a subset of that concept, most commonly *memory safety* but also *bounds safety* and *thread safety*; these concepts represent the absence of *certain kinds* of undefined behavior from the language.

Another important definition of safety is that of *functional safety*: the freedom from unacceptable risk of physical injury or of damage to human health either directly or indirectly (through damage to property or to the environment), which is provided by adequate handling of likely systematic errors, hardware failures, and operational or environmental stress. A related concept is that of *system safety*, which takes a broader view and applies the above notion of safety to the entire system rather than to just one particular component. Note that, unlike language safety (and its subsets, such as memory safety), functional safety and system safety are *statistical* properties; they are typically regulated by policy, and governmental organizations and regulatory bodies typically decide whether software is functionally safe (see [P2026R0]).

These different concepts of "safety" are tied together by another important concept: *correctness*. As defined in [P2900R14], a *correct* program is one that behaves according to its specification and the developer's intent or, in other words, a program that satisfies its plain-language contract.

Like functional safety and system safety and unlike language safety, correctness is a property that only human developers, not computers (programming-language specifications, compilers, tooling, and so on), can know or reason about. The purpose of the Contracts facility proposed in [P2900R14] is to enable the human developer to inject statements about the program's correctness into the program's code to make them visible to and verifiable by the computer.

When used in this way, contract assertions can be leveraged to improve correctness *incrementally*, which no other proposed safety-improving feature can do. The addition of a single contract assertion into an existing C++ application will help improve the application's correctness and stability, and each new assertion will add layers to this benefit. By virtue of being primarily a *correctness feature*, a Contracts facility is, therefore, undoubtedly also a safety feature with regard to functional safety and system safety.

On the other hand, while contract assertions can help with identifying instances of *undefined behavior* (i.e., a possible manifestation of an incorrect program), they are not a safety feature with regard to language safety because they do not add guarantees to the *language* that remove undefined behavior and, in fact, do not change the semantics of the language at all; an existing legacy codebase compiled with a compiler that supports contract assertions will, unless contract assertions are explicitly added and *enforced*, have exactly as much undefined behavior as it did before.

Failure to make the distinction between language safety, functional safety, and correctness is the single biggest obstacle to understanding how Contracts relate to safety. Armed with this deeper understanding, we can now answer the second question that was asked earlier in this section.

One fundamental problem with addressing safety issues is that solutions often focus entirely on the symptoms of the problem rather than on the fundamental causes. CVEs[11] that stem from taking advantage of undefined behavior are, almost tautologically, a result of software bugs that enable those problems to occur; no reasonable developer intends for a program to allow an attacker to take over a system, and any developer with malicious intent is not going to be stopped by any programming language change we could provide. Preventing undefined behavior from allowing exploitation treats the symptom, but a program having a bug in it is the actual problem. The Contracts feature identifies such bugs and provides a way to mitigate them at their source, including a wide variety of options for how Contracts will be applied to programs throughout their entire lifecycle — from development to production systems — while being designed to have as few impediments as possible to its adoption in any C++ codebase of any coding style and any quality.

A more thorough discussion of how the Contracts feature directly addresses safety in C++ can be found in Section 2.3 of [P3204R0] and in [P3500R1]. The critically important distinction between language safety and system safety is discussed in more detail in [P2026R0] (in the context of C++ more generally) as well as in Section 1 of [P3376R0] and particularly in [P3578R0] (in the context of Contracts more specifically). How the Contracts feature fits into the picture of other efforts to address the perception of C++'s safety, such as safety profiles and erroneous behavior, is discussed in more detail in [P3100R1].

Further, [P3100R1] proposes a design direction to address undefined behavior in code *without* explicit contract assertions — and thus also in legacy code — via *implicitly* inserted contract checks. This promising design direction directly addresses language safety but is not being proposed as part of [P2900R14]; instead, it relies on [P2900R14] being approved first and then used as a foundation for future evolution.

2.8 Why Will Contracts for C++ Succeed Where Other Efforts Failed?

As the Contracts proposal — then in revision [P2900R7] — was being presented for the first time to a wider audience beyond SG21, Herb Sutter presented the following question:

> Contracts have been available for decades (e.g., SAL, C# Code Contracts) but aren't widely used in the mainstream. What can make us confident that this one will succeed?

The context of the question itself, as well as a thorough answer, is given in Section 2.1 of [P3204R0]. Here, we provide only a short answer based on the scope of discussion in this paper.

- As demonstrated by the widespread use of assertion facilities in C++ today and the decades of experience we have with such facilities and various tools hooking into them, we know that there is a huge need to address the three use cases that [P2900R14] primarily targets — *documentation*, *runtime checking*, and *static analysis*.
- We further know from this experience that there is a huge interest in overcoming the limitations of such facilities caused by the fact that they are not core-language features.
- [P2900R14] has been designed to overcome these limitations in many specific ways, as listed in Section 2.1 above.

[11] common vulnerabilities and exposures

- [P2900R14] has further been designed to have the widest possible range of applicability and to minimize impediments to its adoption in any C++ codebase of any coding style and any quality.

2.9 Implementation and Deployment Experience

Implementations of earlier Contracts proposals include the Digital Mars implementation of Phase One Contracts and the GCC implementation of C++2a Contracts (see Section 2.6).

Implementations of [P2900R14] are available in GCC and Clang. The GCC implementation is based on the earlier C++2a Contracts version, and the Clang implementation has been developed from scratch. Both implementations are available on Compiler Explorer.[12] Deployment experience was collected from replacing C `assert` in LLVM and dependencies and from replacing the existing hardening, validation, and debugging macros in the libc++ Standard Library implementation with [P2900R14] contract assertions. More information about these efforts and the information collected from them can be found in the latest Implementers Report [P3460R0].

Usage experience was further gained by replacing assertion macros with [P2900R14] contract assertions in the BDE codebase; a report can be found in [P3336R0].

2.10 Implementation-Defined Behavior

[P2900R14] includes a number of aspects that are implementation-defined:

1. The evaluation semantics chosen for each contract assertion evaluation (Section 3.5.5)
2. The number of evaluations of a contract assertion within a sequence (Section 3.5.7)
3. The specific mode of termination used by the *enforce* and *quick-enforce* semantics (Section 3.5.4)
4. The exact behavior of the default contract-violation handler (Section 3.5.9)
5. Whether the contract-violation handler is replaceable (Section 3.5.9)
6. Whether the `contract_violation` object is polymorphic (Section 3.7.3)

The number of instances of implementation-defined behavior has been mentioned in [P2573R0] as a concern. However, all such instances are along boundaries where specific behaviors must be implementation-defined by necessity. Importantly, the situations in which behavior has been made implementation defined are well-known cases for which no single answer is a viable solution for every C++ program and platform, not cases for which we have been unable to find the correct solution nor cases in which the particulars of an implementation alter the contract assertions we will actually write. Implementation-defined properties of a proposal are not a source or indication of confusion; they are an opportunity for compilers and toolchains to provide the abilities that their specific users actually need.

A thorough discussion and motivation of each instance of implementation-defined behavior can be found in the corresponding section referenced in the list above as well as in [P3321R0], which has been reviewed by SG15 (Tooling), and in [P3591R0]. In this section, we provide a brief summary of

[12] GCC implementation: https://godbolt.org/z/7rPxa7TP6
Clang implementation: https://godbolt.org/z/qEo1vGhqM

why each instance of implementation-defined behavior is necessary and which choices the available implementations of [P2900R14] actually make for each one.

The choice of evaluation semantic is implementation-defined because, in general, the Standard does not involve itself in specifying compiler configuration options, and in the past, any attempts to do so have faced significant push-back.

- In both the Clang and GCC implementations of Contracts, the default evaluation semantic for all contract assertions in a translation unit (TU) can be configured with the `-fcontract-evaluation-semantic` option.
- The Clang implementation additionally supports attributes for grouping contracts and a mechanism for controlling grouped contracts (including by namespace) through other command-line flags.
- Both implementations compile contract assertions as part of the function body, so the evaluation semantic of the TU that contains the function definition will take effect.
- Both implementations when a function is inlined will apply the semantics of the TU containing the enclosing function's definition.
- Neither implementation currently does anything to influence the chosen version of an inline function with multiple definitions, so any non-inlined evaluation of such a function will have one of the possible behaviors (though which is unspecified and depends on the linker).

The number of evaluations of a contract assertion within a sequence is implementation-defined because, in general, the effective number will be the result of two different factors.

1. When a call might be doing caller-side and callee-side checking between different TUs, function-contract assertion evaluations may be repeated.
2. When a compiler provides a mechanism (i.e., a command-line option) to ask for repeated evaluations for testing, those will be repeated as requested.

The specific mode of termination is implementation-defined because each of the three allowed modes of termination — `std::terminate`, `std::abort`, and immediate termination — has important use cases and no single choice is capable of covering them all. For the *enforce* evaluation semantic, the Clang implementation of [P2900R14] uses `std::abort` for consistency with the C `assert` macro; for the *quick-enforce* semantic, it uses immediate termination (`__builtin_trap` or `__builtin_verbose_trap`) for `detection_mode::predicate_false` and `std::terminate` for `detection_mode::evaluation_exception` since the latter allows an implementation strategy in which the predicate is evaluated in an implicit `noexcept` context that avoids the overhead of having to surround any potentially throwing predicate evaluation with a `try`/`catch` block. The GCC implementation (currently) uses `std::terminate` for the *enforce* evaluation semantic largely because that is the termination mode that had been specified for C++2a Contracts; for the *quick-enforce* semantic, the GCC implementation behaves similarly to Clang.

The exact behavior of the default contract-violation handler is implementation-defined because the mode and specific formatting for emitting diagnostics differ for various platforms. By allowing implementations the freedom to do what is best in their environments, no users are disenfranchised

or unduly restricted. The current versions of the GCC and Clang default contract-violation handlers output all the values available on the provided `contract_violation` object to `std::cerr`.

Whether the contract-violation handler is replaceable is implementation-defined because replaceability is a key feature of [P2900R14], yet at least one compiler vendor reported that on their platform, providing replaceability poses an unacceptable security risk and they need the ability to disallow making the contract-violation handler replaceable while still being conforming. Both Clang and GCC make the contract-violation handler replaceable, and we expect every other major compiler to follow suit.

Whether `std::contracts::contract_violation` is polymorphic is implementation-defined primarily to allow for the use of `dynamic_cast` to identify whether the provided object is an instance of an implementation-defined subclass of `std::contracts::contract_violation`. Neither of the current implementations currently make this type polymorphic because neither has yet added any implementation-specific additions to `contract_violation` that users might wish to detect.

3 Proposed Design

3.1 Design Principles

The sixteen design principles described in [P2900R14] are largely motivated in that paper itself; in this section, we provide the history and some additional context.

After the experience with earlier Contracts proposals for C++, SG21 expressed interest in working based on design principles when developing the current Contracts proposal, which is captured in the following early SG21 polls.

SG21, Teleconference, 2020-10-06, Poll

We agree that with a statement of priority of principles, and describing the intended programming model, a minimal viable product exists that represents a intersection of possible designs (not merely a deletion of features until there is no controversy).

SF	F	N	A	SA
7	4	2	0	1

SG21, Teleconference, 2020-10-06, Poll

We would like to see a revision of P2182 that includes a statement of priority and the intended programming model.

SF	F	N	A	SA
7	5	1	0	0

However, following these polls, no design principles were initially codified. This lack of action led to situations in which no clear decision-making strategy was available to guide choosing between different mutually exclusive solutions to a problem, none of which were obviously wrong. Examples

of such decisions are how contract assertions should interact with trivial functions (see Section 3.3.3), `noexcept` (see Section 3.6.6), and implicit lambda captures (see Section 3.4.8).

The first set of explicit design principles to guide the design of the Contracts proposal in such cases was proposed in [P2834R1]. This paper was written in the context of build modes; after [P2877R0] was adopted and build modes were removed from the Contracts proposal, a new paper, [P2932R3], proposed an updated set of principles, which were subsequently adopted into the Contracts proposal, starting with revision [P2900R4]. While SG21 never polled any of these design principles separately (we normally poll proposals, not design principles), an extensive discussion of design principles took place at the SG21 meeting in Kona in November 2023. Detailed motivation for the following principles adopted from [P2932R3] can be found in that paper:

- Principle 4: Zero Overhead
- Principle 5: Independence from Chosen Semantic
- Principle 10: Contract Assertions Check a Plain-Language Contract
- Principle 13: Explicitly Define All New Behavior
- Principle 14: Choose Ill-Formed to Enable Flexible Evolution

Principles 13 and 14 were initially one principle but were split as part of a refinement effort in the context of the discussion around undefined behavior in contract predicates (see Section 3.6.1).

Principle 12, Contract Assertions Are Not Flow Control, which implies that contract assertions should not be used for error handling or input validation, is extensively discussed and motivated in [P2053R1], [P1743R0], and [P1744R0].

The remaining design principles guiding the development of the Contracts proposal were codified by gradually being added directly to the Contracts paper as it was developed. Principle 3, Concepts Do Not See Contracts, was added in revision [P2900R4]. Principle 11, Function Contract Assertions Serve Both Caller and Callee, was added in revision [P2900R5]. Principles 15 and 16, the compatibility principles, were added in revision [P2900R7], as well as Principles 2, 6, 7, 8, and 9, all of which originated from the discussion around side effects (see Section 3.5.8), elision, and duplication (see Section 3.5.7) of contract-assertion evaluations.

The last update of the design principles happened in revision [P2900R9] after Oliver Rosten discovered hacks that allow programs to be written that can programmatically detect the presence and semantics of particular contract assertions. This discovery led to the realization that the Contracts Prime Directive and related secondary design principles are not absolute rules that hold in all cases, but rather goals that the Contracts proposal strives to achieve in as many cases as possible. In this revision, we also reordered the principles to better emphasize the primary principles, i.e., Principles 1, 2, and 6.

3.2 Syntax

The syntax for contract assertions proposed in [P2900R14] is the result of over two decades of evolution. The proposed syntax is designed to naturally fit into existing C++; to refrain from overlapping with the design space of other C++ features, such as attributes or lambdas; to be intuitive, lightweight and elegant; and to aid readability by visually separating the different syntactic

parts of a contract assertion. The proposed syntax removes the limitations of previous proposals, such as attribute-like and closure-based syntax, while maintaining full compatibility and extensibility.

A detailed discussion of the motivation for this syntax, how it improves upon previous proposals, and how it can be extended in the future can be found in [P2961R2]. In this section, we summarize the history of how this syntax came about.

As mentioned in Section 2.6, Phase One's D-Like Contracts feature used a D-like syntax with precondition and postcondition blocks (with various minor changes throughout the different revisions), Phase Two's Macro Contracts feature used the syntax for function-like macro invocations (with changing names for the set of proposed macros), and Phase Three's C++2a Contracts feature chose attribute-like syntax.

Attribute-like syntax was first proposed in [N4435], with additional parentheses around the predicate; these parentheses were dropped from [P0246R0] onward. Papers [P0247R0] and [P0380R0] contain motivation for why the attribute-like syntax was chosen at the time; the former paper also mentions why the parentheses were dropped.

In Phase Four, after the Contracts feature was removed from the C++20 Working Paper, SG21 initially adopted the attribute-like syntax from Phase Three (see [P2182R1] and [P2388R4]; the latter paper contains a section motivating the continued use of the attribute-like syntax). However, as work on the Contracts proposal progressed, existing concerns with the attribute-like syntax were raised again, and as a result, SG21 did not have consensus to continue with the same syntax. The concerns raised at that stage included the following.

- The double square brackets were perceived by many as too "heavyweight" and not aesthetically pleasing.
- The attribute-like syntax is treading on the design space of actual C++ attributes, which are a very different feature with different properties, such as ignorability, reorderability, and so on (although the extent to which attributes and contract assertions are really that different in nature was a matter of intense debate).
- Several other syntactic issues specific to function contract specifiers (see Section 3.2.1) were mentioned.

(For a detailed discussion, see [P2487R1], [P2885R3], and [P2961R2].) The attribute-like syntax was pursued further and modifications were proposed in an attempt to address some of the concerns above, such as choosing delimiter tokens different from `[[...]]`, which did not get consensus in SG21.

SG21, Teleconference, 2023-09-07, Poll

We are interested in considering the non-attribute delimiter tokens `[{ ... }]` for the attribute-like Contracts syntax proposed in D2935R1.

SF	F	N	A	SA
1	4	4	10	3

Result: Consensus against

SG21, Teleconference, 2023-09-07, Poll

We are interested in considering the non-attribute delimiter tokens `@(...)` for the attribute-like Contracts syntax proposed in D2935R1.

SF	F	N	A	SA
0	8	5	5	3

Result: No consensus

In parallel, alternative syntaxes were proposed, in particular, the so-called *closure-based syntax* [P2461R1] (which had similarities with earlier syntax proposals in [N1962] and [N4293]) and the so-called *condition-centric syntax* [P2737R0]. Section 10 of [P2461R1] discusses a few more exotic options, such as abbreviated lambda syntax and `pre` blocks (similar to `requires` blocks and the D-like syntax from Phase One), but those additional options were not formally proposed or considered by SG21.

As a result, SG21 was left with three candidate syntaxes: attribute-like, closure-based, and condition-centric. To help SG21 make an informed choice, a list of requirements for a suitable syntax was assembled, and an electronic poll was conducted to determine the relative importance of those requirements; this work is documented in [P2885R3].

Both alternative syntaxes were found to be flawed. The closure-based syntax offered an elegant solution to some of the existing problems but made the unusual choice of placing the predicate expression inside braces. (In C++, statements are placed inside braces, and expressions are placed inside parentheses.) The condition-centric syntax chose the more natural parentheses but also made several other controversial choices and did not consider several important future extensions. Ultimately, both proposals were abandoned.

As a result of these discussions, a new proposal for a so-called *natural syntax* was developed in [P2961R2] and addressed the shortcomings of both the closure-based and the condition-centric syntax proposals. At the November 2023 meeting in Kona, SG21 finally decided to reject all the variants of the attribute-like syntax that were still under consideration at the time and to adopt the natural syntax for the proposed Contracts facility.

SG21, Kona, 2023-11-07, Poll

Adopt the "Attribute-like" syntax, as proposed in P2935R4 Proposal 1-A, for the Contracts MVP.

SF	F	N	A	SA
2	3	14	8	5

Result: Consensus against

SG21, Kona, 2023-11-07, Poll

Adopt the "Attribute-like+Post" syntax, as proposed in P2935R4 Proposal 1-B, for the Contracts MVP.

SF	F	N	A	SA
3	4	11	9	4

Result: Consensus against

SG21, Kona, 2023-11-07, Poll

Adopt the "Attribute-like+Delim" syntax, as proposed in P2935R4 Proposal 1-C, for the Contracts MVP.

SF	F	N	A	SA
2	4	8	9	6

Result: Consensus against

SG21, Kona, 2023-11-07, Poll

Adopt the "Attribute-like+Post+Delim" syntax, as proposed in P2935R4 Proposal 1-D, for the Contracts MVP.

SF	F	N	A	SA
6	7	5	7	5

Result: No consensus

SG21, Kona, 2023-11-07, Poll

Adopt the "Natural" syntax, as proposed in P2961R2, for the Contracts MVP.

SF	F	N	A	SA
14	10	4	4	0

Result: Consensus

A side-by-side comparison of these five polled syntax proposals with more detailed discussion can be found in [P3028R0].

3.2.1 Function Contract Specifiers

Function contract specifiers (i.e., precondition specifiers and postcondition specifiers) are the terms for the *syntactic* constructs `pre(...)` and `post(...)`. Conversely, *function contract assertions*, i.e., precondition assertions and postcondition assertions, are the conceptual entities evaluated when a function is called and when it returns, respectively. Finally, the terms *preconditions* and *postconditions* denote the parts of the *plain-language contract* that are the responsibility of the caller and the callee, respectively.

The motivation for having function contract specifiers as syntactic constructs attached to a function *declaration* — as opposed to assertions inside the definition of a function — is provided above in Sections 2.1 and 2.2. Here, we summarize the *history* of standardizing this feature.

All Contracts proposals except those in Phase Two (Macro Contracts) had syntax for function contract specifiers. Attribute-like syntax initially used `pre` and `post`, respectively, as the context-specific keywords for these specifiers. [N4293] and [N4415] proposed to change this to `expects` and `ensures`, which was adopted for the subsequent merged C++2a Contracts proposals and later the C++20 Working Paper.

[P1344R1] proposed to revert this back to `pre` and `post`. This proposal was adopted by EWG with strong consensus.

EWG, Kona, 2019-02-20, Poll

Change `expects` and `ensures` in the WD definition of contracts to (respectively) `pre` and `post` for C++20.

SF	F	N	A	SA
11	21	8	2	2

Result: Consensus

However, `pre` and `post` never became part of the C++20 Working Paper because the Contracts feature was pulled from it. SG21 retained the `pre` and `post` spelling for the Contracts proposal going forward.

Since precondition and postcondition specifiers are part of a function declaration, a question arises: In which syntactic position these should appear, compared to all the other specifiers on such a declaration?

Attribute-like syntax originally used the syntactic position of attributes that appertain to the function type. One advantage of this position is that we already have established grammar and implementation experience for attributes in this syntactic position. However, one disadvantage is that this position is somewhat awkward since it places function contract assertions before the trailing return type, before specifiers such as `override` and `final` (if we allow `pre` and `post` on virtual functions as a future extension), and before any `requires` clauses. SG21 decided to instead place preconditions and postconditions at the end of the declaration, immediately before the semicolon (or before the beginning of the function body if the declaration is a definition).

SG21, Teleconference, 2023-09-07, Poll

We are interested in placing contract-checking annotations at the end of the function declaration, rather than the position of an attribute appertaining to the function type, for the attribute-like Contracts syntax proposed in D2935R1.

SF	F	N	A	SA
1	11	5	2	1

Result: Consensus

In the end, the natural syntax ([P2961R2]) that had been adopted also placed function contract assertions at the end of the declaration, with the exception that if we allow function contract assertions on virtual functions as future extension, then for pure virtual functions they should be placed before the `= 0` for visual consistency with `= default` (which is grammatically a kind of function body rather than a part of the declarator).

Another issue with attribute-like syntax concerns the syntactic position of the result-name introducer in a postcondition specifier, which is immediately after the `post` and before the colon. This position leads to several syntactic ambiguities with envisioned future extensions, such as labels ([P3400R0]) that appear in the same syntactic position. C++2a Contracts solved this issue by allowing only three such labels — `audit`, `axiom`, and `default` — and specifying that those are never interpreted as result bindings. However, in the future, any number of such labels might be added, which would lead to breaking changes. SG21 reached consensus that a syntax for postcondition specifiers needs to address this issue.

SG21, Teleconference, 2023-10-19, Poll

If we adopt the attribute-like syntax proposed in P2935 for the Contracts MVP, we would like to grammatically resolve any possible syntactic ambiguity between return value names and post-MVP labels in a postcondition.

SF	F	N	A	SA
7	11	0	0	0

Result: Consensus

A similar syntactic ambiguity arises if, as a future extension, we want to introduce postcondition captures and the ability to destructure the result binding; both constructions would use single square brackets in that same syntactic position before the colon. [P2935R4] addressed these issues by introducing an extra colon before and an extra pair of parentheses around the result-name introducer, but this variation of attribute-like syntax did not get consensus (see poll results in Section 3.2). The natural syntax of [P2961R2], which was adopted for the Contracts proposal, addressed both issues by giving each element of the postcondition specifier a distinct, unambiguous syntactic location.

The [P2900R14] syntax for the result-name introducer in a postcondition specifier is discussed in more detail in Section 3.4.3.

One last aspect of the proposed function-contract specifier syntax that deserves discussion is that function contract specifiers can also be applied to lambda expressions, and we have good use cases for such application:

```
constexpr bool add_overflows(int a, int b) {
  return (b > 0 && a > INT_MAX - b) || (b < 0 && a < INT_MIN - b);
}

std::vector<int> vec = { /* ... */ };

auto sum = accumulate(vec.begin(), vec.end(), 0,
  [](int a, int b) pre (!add_overflows(a, b)) {
    return a + b;
  });
```

C++2a Contracts did not allow `pre` and `post` on lambda expressions because it was using attribute syntax, and attributes could not be meaningfully applied to lambdas at the time. According to Section 2.2 of [P0542R5]:

> Contracts attributes (as any other attribute in that syntactical location) appertain to the function type. However, they are not part of the function type. Note that this does not solve the issue of being able to use attributes on lambda expressions (see Core issue 2097). In fact, until that issue is resolved it will not be possible to specify preconditions and postconditions for lambda expressions.

The cited issue was later resolved by adopting [P2173R1] for C++23 and is no longer relevant for [P2900R14] because we are no longer using attribute syntax. However, for a while, the Contracts proposal did not allow `pre` and `post` on lambda expressions for a different reason; Section 3 of [P2388R4] states:

> These features are deferred due to unresolved issues: ... a way to express preconditions and postconditions for lambdas: name lookup is already problematic in lambdas in the face of lambda captures. This problem is pursued in [P2036R1], and until it has been solved we see no point in delaying the minimum contract support proposal.

The cited issue was later resolved by adopting [P2036R3] for C++23 but is, in fact, also irrelevant. The issue solved by [P2036R3] was about name lookup in the trailing return type of a lambda. Name lookup in `pre` and `post` is fundamentally different because identifiers in these predicates should refer to captured entities accessible in the lambda's body, not to entities in the scope containing the declaration of the lambda and which will no longer be available by the time the lambda is called and the preconditions and postconditions are checked. The adopted rule for functions (as-if at the start of the body; see Section 3.4.1) is straightforward and gives us the desired semantics for lambdas.

The necessary grammar for `pre` and `post` on lambdas was provided in the natural syntax paper, [P2961R2], and was approved separately by SG21.

SG21, Teleconference, 2023-12-07, Poll 3

For the Contracts MVP, allow preconditions and postconditions on lambdas with the syntax proposed in P2961R2 and the name lookup rules proposed in P2890R1.

SF	F	N	A	SA
9	9	1	0	0

Result: Consensus

Another noticeable aspect of the syntax for `pre` and `post` in [P2900R14] (which is also found in earlier versions, such as attribute-like syntax) is that precondition specifiers do not have to precede postcondition specifiers but may be freely intermingled with them. Such intermingling is allowed for several reasons. First, precondition and postcondition assertions often come in semantically related pairs or groups, and the user might want to group together, possibly even wrapped by a macro, all assertions that belong to such a group. Second, with the introduction of labels (see [P3400R0]) as a future extension, grouping function contract assertions by label (`symbolic`, `audit`, etc.) can often be more logical than grouping preconditions and postconditions together.

For a discussion of the name lookup rules referred to in the above poll, see Section 3.4.1. For a discussion of the interaction between contract assertions and implicit lambda captures, see Section 3.4.8.

During CWG wording review of [P2900R12], the question came up whether precondition and postcondition assertions should be allowed to be attached to `main`, considering that `main` is not called from another part of the program the way other functions are, and returning from `main` is specified differently as well. The result of the discussion was that `pre` and `post` on main seems useful — for example, to assert conditions on static objects initialized before `main` is called, or to assert a condition on the value being returned to the host environment — and nothing in the specification prohibits that from working as expected. The same reasoning applies to other "special" functions such as the contract-violation handler (see Section 3.5.9).

3.2.2 Assertion Statement

The motivation for why, if we have precondition and postcondition specifiers on function declarations, we still also need contract statements that can be used inside a function definition is provided above in Section 2.2. Here, we summarize the history of standardizing this feature.

Phase One and Phase Two Contracts facilities did not have a syntax for assertion statements distinct from precondition and postcondition specifiers. The idea for an assertion statement that can be placed inside a function body, like existing macro `assert`, and unlike precondition and postcondition specifiers, which are placed on function declaration, arose in Phase Three when the attribute-like syntax was developed. In this syntax, the assertion statement was always spelled as `[[assert:` *expr*`]]`.

For the natural syntax of [P2961R2] that was later adopted for the Contracts proposal, the problem arose that an assertion statement would have to be introduced by a *full* keyword (unlike `pre` and `post`, which can be contextual keywords) to disambiguate the assertion statement from a function call. The natural keyword — and the keyword used in every other programming language for this purpose — is `assert`, but in C++, this keyword is already taken by macro `assert`. A desirable fix seemed to be to retrofit macro `assert` into the new Contracts feature by using `assert` as the keyword for assertion statements, since after all, both have the same default behavior, print a message, and terminate, but after much discussion, SG21 realized that this solution is not possible because macro `assert` is controlled by `NDEBUG` and has several other important differences and thus reusing the `assert` keyword would mean either a breaking change to macro `assert` or a situation in which `assert(x)` means different things in different contexts, neither of which is a viable option.

SG21, Teleconference, 2023-09-21, Poll

If we adopt the syntax in P2961R0 for the Contracts MVP, we should use the keyword `assert` for contract assertions, replacing macro `assert`.

SF	F	N	A	SA
0	2	3	6	5

Result: Consensus against

Thus, the search for a new keyword that is not `assert` began. Over forty candidate keywords were proposed, ten design goals were formulated, and an analysis was conducted of which candidate keywords best meet the formulated design goals. This work is documented in [P2961R2], Section 5.3. Two candidate keywords, `assertexpr` and `contract_assert`, emerged as the winners, and SG21 chose the latter by poll.

SG21, Teleconference, 2023-10-26, Poll

In case we choose to adopt the "natural syntax" for Contracts as proposed in P2961, we should use `assertexpr` as the keyword for assertions.

SF	F	N	A	SA
1	3	8	7	0

Result: No consensus

SG21, Teleconference, 2023-10-26, Poll

In case we choose to adopt the "natural syntax" for Contracts as proposed in P2961, we should use `contract_assert` as the keyword for assertions.

SF	F	N	A	SA
9	11	0	0	1

Result: Consensus

The discussion that led to these poll results revealed that while both candidate keywords were seen as a good fit for C++ when considering C++ in isolation, `contract_assert` was a better candidate for consistency with other programming languages since it is more visibly a variation of `assert`, which would be the expected keyword.

Another design question regarding the syntax for assertion statements is whether they should be statements or expressions, considering that the existing `assert` macro can be an expression. Making `contract_assert` a statement means that, in some situations, the latter cannot be used as a drop-in replacement for the former, and workarounds are required such as wrapping the `contract_assert` into an immediately invoked lambda (see Section 3.6.6 of [P2900R14] for an example). Nevertheless, assertion statements were statements throughout all proposals that used attribute-like syntax. The natural syntax [P2961R2] offered an opportunity to make `contract_assert` an expression, which gained consensus in SG21.

SG21, Teleconference, 2023-10-19, Poll

Should the Contracts MVP allow assertions to be expressions as opposed to just statements?

SF	F	N	A	SA
4	7	3	3	0

Result: Consensus

However, we later realized that if `contract_assert` is an expression, then we need to answer the question of whether `contract_assert` is a potentially throwing expression, i.e., whether

`noexcept(contract_assert(x))` is `true` or `false`, which is fairly contentious and does not have an obvious answer (see Section 3.6.6; for a detailed discussion, see [P2969R0], Section 3.7 of [P2932R2], [P3113R0], and [P3114R0]).

The status quo is that neither making `noexcept(contract_assert(x))` return `true` nor making it return `false` nor making asking the question ill-formed (which could be accomplished in several subtly different ways; the options and their tradeoffs are listed in [P2969R0]) could get consensus in SG21. This leaves reverting `contract_assert` to being a statement as the only viable option, thus avoiding the question altogether, which is what SG21 decided to do for the Contracts proposal.

SG21, Teleconference, 2024-02-01, Poll 1

For the Contracts MVP, make `contract_assert` a potentially-throwing expression (P3113R0 Option 1, P2969R0 Option 3.1).

SF	F	N	A	SA
3	1	1	3	4

Result: No consensus

SG21, Teleconference, 2024-02-01, Poll 2

For the Contracts MVP, clarify that `contract_assert` is not a potentially-throwing expression (P3113R0 Option 2, P2969R0 Option 3.2).

SF	F	N	A	SA
2	2	3	2	4

Result: No consensus

SG21, Teleconference, 2024-02-01, Poll 3

For the Contracts MVP, when determining if a set of expressions is potentially-throwing, `contract_assert` is not considered; if there are no expressions other than `contract_assert`, the program is ill-formed (P3113R0 Option 3, P2932R2 Proposal 7A).

SF	F	N	A	SA
2	6	0	0	5

Result: No consensus

SG21, Teleconference, 2024-02-01, Poll 4

For the Contracts MVP, make `contract_assert` a statement rather than an expression (P3113R0 Option 6a, P2969R0 Option 3.5.1).

SF	F	N	A	SA
7	5	1	1	0

Result: Consensus

SG21, Teleconference, 2024-02-01, Poll 5

For the Contracts MVP, `contract_assert` is neither potentially-throwing nor not potentially-throwing; if a contract_assert appears as a subexpression of the operand of `noexcept` or while deducing an exception specification, and no other subexpression is potentially-throwing, the program is ill-formed (P3113R0 Option 6c, P2969R0 Option 3.5.3, P2932R2 proposal 7B).

SF	F	N	A	SA
0	5	5	2	1

Result: No consensus

SG21, Teleconference, 2024-02-01, Poll 6

For the Contracts MVP, contract-violation handlers shall be `noexcept`; clarify that `contract_assert` is not a potentially-throwing expression. (P3113R0 Option 8, P2969R0 Option 3.7).

SF	F	N	A	SA
1	1	3	3	6

Result: Consensus against

The last poll simultaneously reconfirmed that SG21 has consensus on allowing throwing contract-violation handlers (see Section 3.6.6).

3.2.3 Attributes for Contract Assertions

Attributes appertaining to contract assertions are useful but impractical with attribute-like syntax since C++ does not have the notion of attributes appertaining to attributes or of nested attributes. However, with the change to natural syntax ([P2961R2]), these ideas became a possibility.

The Contracts proposal addresses attributes that appertain, in three distinct fashions, to contract assertions (or parts of them).

1. Contract-assertion–specific attributes can appertain to a contract assertion itself and may appear in the space after the `pre`, `post`, or `contract_assert` and in the parenthesized conditional expression.
2. Attributes, such as `[[likely]]` and `[[unlikely]]`, that can appertain to any statement can also appertain to an assertion statement using `contract_assert`. To be consistent with all other statements, such attributes should be placed at the beginning of the statement.
3. Attributes, such as `[[deprecated]]`, that can appertain to any declared names can also appertain to the result binding in a postcondition specifier. As with other declarations of new names, the attribute should be placed after the result-name introducer and prior to the subsequent colon (`:`).

The first kind, Contracts-specific attributes, can be very useful for vendor extensions to provide any sort of additional functionality to a contract assertion that is not covered by [P2900R14]. This flexibility ranges from platform-specific needs, such as adding information for a static analysis

tool consuming these assertions, to early implementation and experimentation with planned future extensions, such as labels (see [P3400R0]).

As an anecdote, when the Contracts proposal was first shown to a particular implementer of the C++ Standard Library who was interested in using contracts instead of their current internal assertion macros, their first question was how they could specify a custom human-readable error message for a contract violation. When they learned that this feature was not part of the Contracts proposal, their second question was where the syntactic position for attributes was on a contract assertion so that they could add that feature to their compiler implementation.

The main motivation for the second and third kinds is consistency with the rest of the C++ language.

Detailed motivation for the first two kinds of attributes, including a thorough analysis of appropriate syntactic positions, can be found in [P3088R1]; motivation for the third kind can be found in [P3167R0].

All three proposals were approved by SG21 for the Contracts proposal.

SG21, Teleconference, 2024-02-15, Poll 1

Allow contracts-specific attributes to appertain to `pre`, `post`, and `contract_assert`, using the infix position, as proposed by P3088R1 Section 4.1.

SF	F	N	A	SA
2	10	2	0	1

Result: Consensus

SG21, Teleconference, 2024-02-15, Poll 2

Allow attributes such as `[[likely]]` and `[[unlikely]]` that can appertain to a statement to also appertain to `contact_assert`, using the prefix position, as proposed by P3088R1 Section 4.2.

SF	F	N	A	SA
6	6	3	1	0

Result: Consensus

SG21, Teleconference, 2024-02-29, Poll 2

For the Contracts MVP, allow attributes for the result name in a postcondition assertion as proposed in P3167R0.

SF	F	N	A	SA
2	7	4	1	0

Result: Consensus

In addition, the existing `[[deprecated]]` attribute was given explicit allowance to appertain to the result binding in a postcondition assertion, to make it consistent with other introduced names.

SG21, Teleconference, 2024-02-29, Poll

For the Contracts MVP, allow `[[maybe_unused]]` to appertain to the result name in a postcondition assertion as proposed in P3167R0.

SF	F	N	A	SA
3	5	7	0	0

Result: Consensus

3.3 Syntactic Restrictions

3.3.1 Multiple Declarations

The purpose of the rule that the function-contract-assertion sequence of a function has to be present on the *first* declaration of that function in the given translation unit is to ensure that every place in the program agrees on what the function-contract-assertion sequence of that function is. This common understanding enables compilers to generate code for checks either caller-side or callee-side. Tools and humans are also able to reason consistently about the contracts that apply to a function invocation without any need to see the function body.

We *allow* the function-contract-assertion sequence of a function to be repeated on a subsequent function declaration because it might aid readability and maintainability; see [P3066R0] for discussion. For example, a precondition predicate might use a member variable that is then also used in the function body. Having that precondition assertion *on* the function body rather than only on the first definition (which might be far away in a header somewhere) is arguably more user friendly.

At the same time, we do not *require* the function-contract-assertion sequence of a function to be repeated on every declaration of that function because compilers and other tools do not need it and because unnecessarily repeating a long sequence of function contract assertion could be burdensome for the user.

Allowing repetition of the function-contract-assertion sequence requires us to have a precise rule for when two instances of such a sequence are considered to be *the same*. Since we must ensure that every place in the program agrees on what the function-contract-assertion sequence of that function is, we make it ill-formed for two instances of such a sequence on the same function to not be the same, but we cannot require a diagnostic if the two declarations are in different translation units.

The rule that redeclarations of precondition or postcondition assertions in the same translation unit cannot contain lambda expressions stems from the fact that implementing a correct "sameness" check for such redeclarations would require unreasonable heroics on both the specification side and the implementation side (see Section 2 of [P3520R0] for a detailed discussion). In essence, when two lambda expressions occur in a C++ program, they are always treated as distinct entities introducing distinct closure types, even if the lambda expressions themselves are token identical; changing that behavior is highly nontrivial. Moreover, if parameter names are involved, the needed mapping being applied to lambdas within a predicate might result in two lambdas that are not token identical but would still have to be considered identical under the rule above, so token-comparing the lambdas is insufficient but becomes necessary to fully parse them. In at least one major compiler, this parsing cannot be done currently without creating a distinct closure type for each lambda.

The rules around first and nonfirst declarations changed several times during the history of Contracts proposals for C++. Phase One's Contracts proposals did not discuss the rules for function contract assertions on multiple declarations of the same function. Phase Two's Contracts proposals had no function contract assertions but only assertion macros inside function bodies.

The first explicit set of rules for function contact assertions on declarations was proposed in Phase Three in [N4415] and [P0380R0]: When there are multiple declarations of a function with function contract assertions, the sequence of function contract assertions must be repeated on every declaration in an ODR-identical way (same token sequence; names refer to same entities); otherwise, the program is ill-formed, no diagnostic required (IFNDR).

In [P0380R1], these rules were relaxed: The sequence of function contract assertions must be placed on a first declaration[13] but can be omitted from a nonfirst declaration. (When not omitted, it must still be ODR identical.) This relaxed rule was retained in Phase Four and appears in [P2388R0]. The next revision of this proposal, [P2388R1], added a new rule that when the function-contract-assertion sequence is inconsistent across translation units, the behavior is undefined; [P2388R2] changed this situation back to IFNDR.

The first revision, [P2900R0], of the current Contracts paper took yet another approach and proposed that function contract assertions can appear *only* on a first declaration, not on any redeclarations of a function. This approach has several drawbacks but was deemed necessary at the time because how the determination of whether two sequences of function contract assertions are "the same" could even be made was unclear. Such a determination would not have to be made for different first declarations of the same function in different translation units since any inconsistency between those declarations can be made IFNDR, but it would have to be made if we allow repetition of the sequence of function contract assertions on a redeclaration (i.e., a declaration in the same translation unit, from which the first declaration is reachable). The authors recall that SG21 discussed and polled this approach explicitly. However, the minutes from this particular telecon appear to have been lost, so we cannot reproduce the relevant poll result here.

Later, a rule to make this determination was developed in [P2932R3] and was offered in Proposal 6.2 of that paper; however, Proposal 6.1 of the same paper also suggested retaining the rule that function contract assertions can appear only on a first declaration, motivated by a desire to keep the functionality set minimal and avoid feature creep.

The counterproposal [P3066R0] argued that since we now have a rule to determine whether two sequences of function contract assertions are the same, that rule can be used to allow repeating the sequence on redeclarations, which is undoubtedly useful and does not actually constitute feature creep because this feature had already been part of an earlier version of the Contracts proposal. This counterproposal, along with the rule for determining sameness itself, was adopted by SG21.

[13] Note that the term *first declaration* is never defined in the C++ Standard despite being used in about a dozen places. The definition proposed in [P2900R14], which we believe matches the intended meaning of the term in the C++ Standard and in earlier Contracts papers, is that a first declaration is a function from which no other declaration is reachable. Unlike a definition based on lexical order, this definition also works with modules and with friend declarations inside templates. Note that this definition implies that each translation unit has its own first declaration of a function, and those do not have to be the same declaration. Therefore, a function can have multiple first declarations, which is somewhat unintuitive.

SG21, Teleconference, 2024-01-25, Poll 2

For the Contracts MVP, adopt the rules for sameness of contract assertions as proposed in P2932R3 Proposal 6.2.

SF	F	N	A	SA
8	8	0	0	0

Result: Consensus

SG21, Teleconference, 2024-01-25, Poll 3A

For the Contracts MVP, a function declaration that is not a first declaration shall have either no precondition or postcondition specifiers, or the same precondition or postcondition specifiers as the first declaration, as proposed in P3066R0.

SF	F	N	A	SA
7	8	2	0	0

Result: Consensus

SG21, Teleconference, 2024-01-25, Poll 3B

For the Contracts MVP, a function declaration that is not a first declaration shall not have precondition or postcondition specifiers, as proposed in P2932R3 Proposal 6.1.

SF	F	N	A	SA
3	4	4	5	0

Result: No consensus

In the later stages of Phase Three, [P1320R2] proposed yet another direction that would allow, for *member* functions only, function contract assertions on redeclarations while *omitting* them from the first declaration. Thus, the function contract assertions of a member function could effectively become an implementation detail of that function. The paper provides the motivation for why this ability is necessary and useful, along with technical limitations that this approach implies (such as where and under what circumstances the compiler can generate code for contract checks) and implementation experience in GCC. However, this direction did not get consensus in EWG at the time.

EWG, Kona, 2019-02-19, Poll

P1320R1 (Allowing contract predicates on non-first declarations) for C++20

SF	F	N	A	SA
1	4	12	6	2

Result: No consensus

More recently, the same idea was brought up again during SG21 discussions as a useful extension and is discussed in more detail in Section 2.2 of [P2755R1]; however, this feature was ultimately not considered for [P2900R14] and is instead planned as a possible future extension.

Section 1.4 of [P3483R1] first considered the problem that allowing a redeclaration of precondition and postcondition assertions that contain lambda expressions in their predicate also requires an answer to the question of whether and when those lambda expressions are "the same." The paper proposed that in such cases, we should apply the same rules for what is or is not an ODR violation as we already do for any other contract predicates. This proposal was approved by SG21 alongside the other clarifications proposed in [P3483R1]. (For the poll result, see Section 3.4.3.)

However, CWG's wording review of [P2900R11] revealed that such a design does not work for the reasons stated further above in this section. Section 2 of [P3520R0] proposed the new rule that redeclarations of precondition and postcondition assertions in the same translation unit cannot contain lambda expressions as the preferred solution to that problem. Alternative solutions were considered and are also described in Section 2 of [P3520R0]. The preferred solution was subsequently adopted by SG21 and EWG.

SG21, Wrocław, 2024-11-22, Poll 2

Disallow lambdas inside pre/post on redeclarations as proposed in P3520R0 Section 2.

SF	F	N	A	SA
5	10	1	0	0

Result: Consensus

EWG, Wrocław, 2024-11-22, Poll

P3520r0 contracts, EWG supports the paper's suggested fix for item 2, which adds a note clarifying that lambda-expressions in pre and post conditions cannot be redeclared in the same translation unit.

SF	F	N	A	SA
13	22	5	2	0

Result: Consensus

3.3.2 Virtual Functions

Runtime polymorphism and dynamic dispatch through virtual function calls is an important paradigm in C++ and many other programming languages. Those programming languages that offer a Contracts facility as a core-language feature, such as Eiffel, D, and Ada, all allow specifying function contract assertions on virtual functions and integrate the runtime checking of those assertions into their runtime polymorphism facilities. Section 2 of [P3097R0] provides an overview of the design space and a description of the functionality in these languages.

In general, Eiffel and D follow the so-called *classic model* of Design by Contract. The function contract assertion on an overridden function is automatically inherited by the overriding function; the overriding function can define its own function contract assertions, which are applied in addition to those of the overridden function, according to the basic principle that the preconditions of the overridden function can only be widened and the postconditions can only be narrowed, which is achieved by using a conjunction (via OR) of the precondition assertions and a disjunction (via AND) of the postcondition assertions of the overriding function and all functions in the inheritance

hierarchy overridden by it. Ada follows a more complex model: A variation of the classic model is realized through so-called *class-wide* function contract assertions (spelled `Pre'Class` and `Post'Class`), but the default is actually the so-called *specific* function contract assertions (spelled `Pre` and `Post`), which are checked when *that* function is called, regardless of whether it is a virtual function or whether it is overriding or overridden by other functions.

However, [P2900R14] does not follow this classic model for several reasons. One reason is that in C++, automatic inheritance of contract assertions leads to several problems, such as remote code breakage (as discussed in Section 3.4 of [P2932R3]). Another reason is that the classic model precludes some important use cases of virtual functions in C++, which go beyond the object-oriented paradigm normally used in other programming languages (as discussed in much detail in [P3097R0]).

In particular, the classic model does not leave room for an overriding function to violate the postconditions of the overridden function if it is called outside the domain of that overridden function, even though this valid use of Contracts on virtual functions does not violate the substitution principle; the classic model does not leave room for an overriding function whose substitutability for the overridden function is a dynamic rather than a static property (e.g., "Is Optimus Prime a truck?") and breaks down completely for multiple inheritance, which does not exist in Ada, D, or Eiffel but is widely used in C++. [P3097R0] contains code examples for all those cases.

In addition, using a conjunction (via OR) of precondition assertions, as in Eiffel, D, and Ada, makes no sense in the execution model of [P2900R14]. Fundamentally, contract assertions do not produce a boolean logic and identify only cases in which a violation occurs.

Treating a sequence of assertions that failed to detect a violation as confirmation that a contract was actually satisfied is a fallacy; contract assertions might be only observed or ignored, and perhaps not all aspects of the underlying plain-language contract have been encoded as contract assertions. Due to this fallacy, evaluating one assertion sequence and seeing no violation is not a sound reason to treat the conjunction of that sequence with another one as if it couldn't be a violation.

A violated contract assertion is a sign that something is wrong, which *might* be the direct caller or *might* be a more fundamental systemic failure. The Contracts feature makes a fundamental promise that the contract-violation handler will be called as soon as a violation is detected, giving program owners the option to fail fast on any systemic failure. A conjunction will, when one operand of the conjunction is found to be false, proceed to determine if the other operand is also false. Attempting to apply the same logic to contract assertions would require delaying the detection of a violation or somehow rolling back an earlier violation, neither of which is remotely viable or wise in the model of [P2900R14].

Alternatives to the classic model exist and seem to be a better fit for C++. One particularly promising model, the caller/callee model, was developed in [P3097R0] and initially added to the Contracts proposal in revision [P2900R8]. However, due to growing concerns over a lack of experience with deploying and using this model as well as evidence of existing use cases for `pre` and `post` on virtual functions that are *not* covered by this model, EWG ultimately decided to ship [P2900R14] without support for `pre` and `post` on virtual functions and instead to add such support as a later extension once we are more confident about the correct model to use. While virtual functions are an important paradigm in C++, shipping Contracts for C++ without supporting them still provides significant value to users and is much better than not shipping Contracts for C++ at all.

The history of the decision to ship without initial support for virtual functions, like many other design decisions that ultimately led to [P2900R14], is long and eventful.

The first Phase One proposal [N1613] was directly inspired by the D language and, therefore, proposed the same design as in D: An overriding function automatically inherits its function contract assertions from the overridden function, but these contract assertions too can be overridden, with the constraint that the preconditions cannot be stronger and the postconditions cannot be weaker, implemented by using a logical conjunction (via OR) of the preconditions and disjunction (via AND) of the postconditions across the hierarchy. In the next revision of that paper, [N1669], this design was retained for precondition assertions, but postconditions had to be the same as in the overridden function, with the rationale that weakening postconditions is "useless in practice since the user cannot take advantage of the weaker precondition by performing a downcast."

Phase Two Contracts did not offer function contract assertions and, therefore, did not offer any integration of Contracts with runtime polymorphism.

The importance of integrating Contracts with runtime polymorphism in C++ was recognized early in Phase Three: The basic principle was again stated to be that preconditions cannot be strengthened and postconditions cannot be weakened in an overriding function; see discussion in [N4110] and [P0147R0]. A different opinion was expressed in [P0247R0], which stated, "Inheriting checks on virtual functions, or requiring that checks on base class virtual interfaces match overriders does not work." The paper mentioned that allowing an overriding function to both widen and narrow the preconditions of the overridden function, breaking with the established paradigm, might make sense because not everyone will want to call a function through the same interface.

[N4415] and [P0287R0] proposed that an overriding function should automatically inherit the sequence of function contract assertions from the overridden function. This sequence can be optionally repeated or omitted; if it is repeated, it must be ODR identical (similar to the rule used for redeclarations of the same function). The ability to strengthen or weaken the contract assertions on an overriding function was omitted from these proposals "out of simplicity concerns" (as described in Section 2.3 of [P0287R0]).

[P0380R0] made this rule even more restrictive by requiring that the sequence be repeated on an overriding function. The motivation given was that this rule could be relaxed later if needed but should not be done at that time "because the complexity is not worth it." In the next revision, [P0380R1], this restriction was again rolled back, and omitting the sequence on an overriding function was allowed.

This rule was retained throughout Phase Three (including the version that went into the C++20 Working Paper). None of the Phase Three proposals allowed an overriding function to have a sequence of function contract assertions different from the overridden function. This rule also means that, for multiple inheritance, the corresponding member function has the same sequence in all classes involved in the hierarchy. The definition of "the same" used here for overrides also included the addition of an IFNDR case if "the corresponding conditions will always evaluate to the same value" (according to the wording in [P0542R5]) and generally avoided clear answers to the questions of how two conditions should be treated as equivalent when they are applied to objects of different types.

In Phase Four, the Contracts proposal continued to flip-flop between these two designs: [P2388R4] made repeating the sequence on an overriding function optional, and [P2521R5] made it mandatory, both papers offering some discussion on this topic. In particular, [P2388R4] contains the statement that doing anything other than inheriting the contract is "conceptually wrong."

The various disagreements about and different understandings of how contracts on virtual function overrides should behave led to [P2954R0], which proposed that repeating the sequence on an overriding function was neither optional nor required but actually ill-formed and that the contract was always inherited implicitly, which was intended as "a simple solution that makes the uncertain cases ill-formed, and makes well-formed the cases that have no future compatibility risk." More detailed motivation and discussion can be found in that paper. This proposal was approved by SG21.

SG21, Teleconference, 2023-08-10, Poll 1

For the Contracts MVP, an overriding function shall not specify preconditions or postconditions, and inherits those of the overridden function.

SF	F	N	A	SA
4	5	2	2	0

Result: Consensus

SG21, Teleconference, 2023-08-10, Poll 2

For the Contracts MVP, if a function overrides more than one function, neither the overriding function nor any of the overridden functions shall specify preconditions or postconditions.

SF	F	N	A	SA
6	7	0	1	0

Result: Consensus

However, after more investigation, implicit inheritance of the contract was found to be problematic; more detailed discussion can be found in Section 3.4 of [P2932R3]. SG21 decided that, given the many concerns and disagreements over Contracts on virtual functions, consensus on a comprehensive solution could be deferred to a future extension, and virtual function support was thus removed entirely from the proposal.

SG21, Teleconference, 2024-01-25, Poll 1

For the Contracts MVP, make it ill-formed for a virtual function to have precondition or postcondition specifiers as proposed in P2932R3 Proposal 4.

SF	F	N	A	SA
9	7	1	1	0

Result: Consensus

Following this decision, the Contracts proposal was forwarded to EWG and LEWG without support for `pre` and `post` on virtual functions. However, in [P3173R0], a major compiler vendor stated that, given the importance and persuasive use of virtual functions in C++, "a Contracts facility that fails

to adequately support efficient use with virtual functions ... is woefully inadequate and unready for prime use." When [P3173R0] was discussed at the WG21 meeting in Tokyo (March 2023), EWG took a poll on whether virtual function support must be included in the first version of the Contracts facility that we ship.

EWG, Tokyo, 2024-03-20, Poll 3

P2900R6 Contracts should specify contracts on virtual functions in its Minimal Viable Proposal.

SF	F	N	A	SA
8	15	10	13	5

Though this poll did not result in consensus either way, it led to the realization that a Contracts proposal is more likely to be adopted for the C++ Standard if it has support for `pre` and `post` on virtual functions and renewed efforts to specify such a feature. Two opposing models emerged.

The first model has been in development since 2019 and was first published in [P3097R0], which provided extensive rationale and motivating examples. [P3165R0] provided additional motivation for the same proposal. This model separates function contract assertions into caller-facing and callee-facing function contract assertions. It proposes that for any function call, both sets of contract assertions should be checked in a particular sequence (see Figure 1).

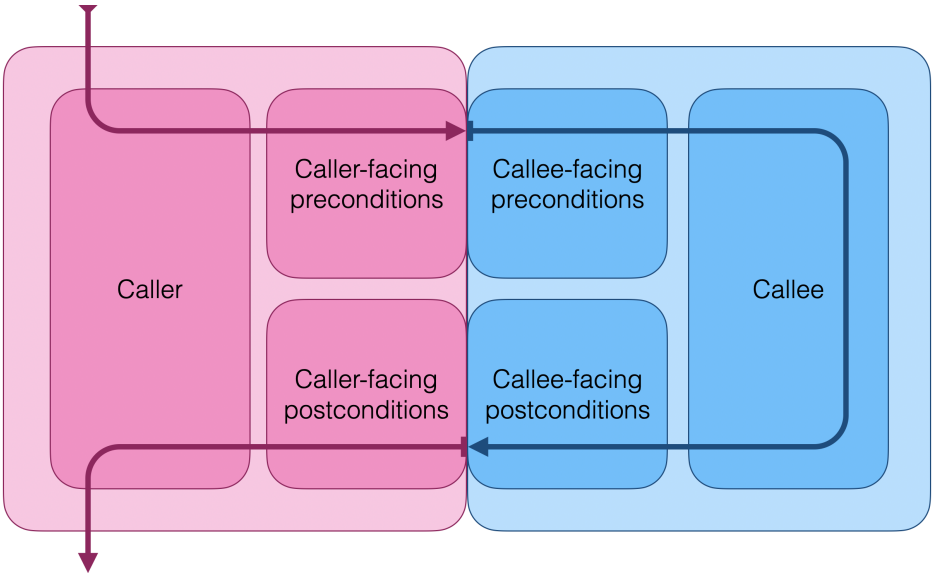

Figure 1: Evaluation sequence of caller-facing and callee-facing function contract assertions

The callee-facing function contract assertions of a function call are *always* the function contract assertions attached to the function whose body is being evaluated. The caller-facing function contract assertions of a function call, on the other hand, are determined based on how the function is invoked. For nonvirtual function calls, the caller-facing assertions are identical to the callee-facing ones; for calls through a function pointer, the set of caller-facing assertions is empty; and for virtual function

calls, the caller-facing assertions are those on the statically called function (while the callee-facing assertions are the ones on the final overrider selected by virtual dispatch).

Notably, this model treats the function contract assertions on every involved function as independent from each other. This model is an explicit departure from previous proposals based on inheriting the contract assertions from an overridden function and from existing practice in Eiffel, D, and Ada. [P3097R0] and [P3165R0] argued that the caller-facing and callee-facing model is better suited for how virtual functions work in C++ and supports many important use cases that the other designs do not, including meaningful support for multiple inheritance.

In this model, depending on how a virtual function is used, its function contract assertions can act as the caller-facing contracts, the callee-facing contracts, or both. An unpublished draft revision of [P3097R0], D3097R1, proposed an extension whereby a function contract assertion on a virtual function can optionally be labelled as an `interface` contract or an `implementation` contract, meaning that it is checked only if it acts as a caller-facing contract or a callee-facing contract, respectively. This extension to the basic caller-facing and callee-facing model enables some additional specific use cases that the base model does not directly address. (Workarounds are available but might require additional code to be written or existing code to be modified to accommodate the exact sequence of contract checks desired for those use cases.)

The second, competing model was proposed in [P3169R0]. In this model, function contract assertions on virtual functions are inherited by overriding functions, with constraints that preconditions should not be tightened and postconditions should not be weakened, thereby following the classic model, which is existing practice in Eiffel, D, and Ada (with some adjustments for preconditions).

In a first round, SG21 reviewed all the above proposals and took the following directional polls.

SG21, Teleconference, 2024-06-06, Poll 1

SG21 would like to spend more time considering, for the Contracts MVP, `pre` and `post` for virtual functions as proposed in the base proposal of D3097R1 and in P3165R0: a virtual function call first evaluates the precondition assertions of the statically called function, then those of the final overrider, then executes the body of the final overrider, then evaluates the postconditions of the final overrider, and finally those of the statically called function; performing a virtual function call through a pointer to member function checks only the precondition and postcondition assertions of the final overrider.

SF	F	N	A	SA
4	8	0	0	4

Result: Consensus (but concerns raised regarding the concrete approach should be addressed)

SG21, Teleconference, 2024-06-06, Poll 2

SG21 would like to spend more time considering, for the Contracts MVP, `pre` and `post` for virtual functions as proposed in P3169R0: a virtual function call first evaluates the precondition assertions of the statically called function, then executes the body of the final overrider, and finally evaluates the postconditions of all overridden functions.

SF	F	N	A	SA
2	0	1	8	5

Result: Consensus against

SG21, Teleconference, 2024-06-06, Poll 3

SG21 would like to spend more time considering, for the Contracts MVP, adding explicit `interface` and `implementation` contract assertions for virtual functions as proposed in the extended proposal of D3097R1.

SF	F	N	A	SA
2	7	3	0	3

Result: Consensus

In a second round of discussion, SG21 looked more closely at the different possible models and their semantics and at some concrete concerns over the base model and the extensions proposed in D3097R1 and took the following directional polls.

SG21, Teleconference, 2024-06-13, Poll 1

The Contracts MVP should encourage the principle that you cannot narrow the precondition assertions and you cannot widen the postcondition assertions of an overridden function in an overriding function.

SF	F	N	A	SA
2	0	0	8	5

Result: Consensus against

SG21, Teleconference, 2024-06-13, Poll 2

In the Contracts MVP, for a virtual function call, the set of precondition and postcondition assertions that will be checked for a particular overriding function should be independent of how (through which statically called function) that function is called.

SF	F	N	A	SA
1	1	1	3	10

Result: Consensus against

SG21, Teleconference, 2024-06-13, Poll 3

The Contracts MVP should offer some mechanism to inherit the precondition and postcondition assertions of an overridden function into functions overriding it.

SF	F	N	A	SA
0	2	1	11	1

Result: Consensus against

SG21, Teleconference, 2024-06-13, Poll 4

Contracts should allow `pre... (x)` and `post... (x)` on a virtual function where `...` are some syntactic markers to explicitly specify them as interface contracts (checked when that function is the statically called function of a virtual function call), implementation contracts (checked when that function's body is executed), or both simultaneously.

SF	F	N	A	SA
1	11	2	0	1

Result: Consensus

SG21, Teleconference, 2024-06-13, Poll 5

Contracts should allow unadorned `pre (x)` and `post (x)` on a virtual function meaning contract assertions that are simultaneously interface contracts and implementation contracts.

SF	F	N	A	SA
5	4	3	1	3

Result: Weak consensus

SG21, Teleconference, 2024-06-13, Poll 6

Contracts should allow unadorned `pre (x)` and `post (x)` on a virtual function meaning contract assertions that are implementation contracts.

SF	F	N	A	SA
1	1	7	5	2

A third round of SG21 review followed at the WG21 meeting in St. Louis (June 2024). At this meeting, SG21 decided to forward the base model of D3097R1, which is exactly equivalent to the proposal in [P3097R0], to EWG, while postponing further discussion of the `interface` and `implementation` qualifiers for future extensions since they do not seem to be essential for a first version of a C++ Contracts facility.

SG21, St. Louis, 2024-06-27, Poll 1

Forward `pre`/`post` on virtual functions, without additional qualifiers, as proposed in the "Base Proposal" of D3097R1, to EWG, as an extension of P2900, with C++26 as the recommended ship vehicle.

SF	F	N	A	SA
10	3	0	2	0

Result: Consensus

SG21, St. Louis, 2024-06-27, Poll 2

Forward `pre`/`post` on virtual functions with additional qualifiers, as proposed in the "Extended Proposal" of D3097R1, except that the qualifier `interface` is renamed to `indirect`, to EWG as an extension of P2900, with C++26 as the recommended ship vehicle.

SF	F	N	A	SA
0	1	4	7	3

Result: Consensus against

SG21, St. Louis, 2024-06-27, Poll 3

We want to spend more time considering the "Extended Proposal" of D3097R1, in the MVP timeframe.

SF	F	N	A	SA
3	3	5	5	0

Result: No consensus

SG21, St. Louis, 2024-06-27, Poll 4

We want to spend more time considering the "Extended Proposal" of D3097R1, as a post-MVP extension.

SF	F	N	A	SA
7	6	3	0	0

Result: Consensus

After a presentation of [P3344R0] to EWG at the same meeting, the caller/callee model for `pre` and `post` on virtual functions as proposed in [P3097R0] was approved by EWG.

EWG, St. Louis, 2024-06-28, Poll 1

P3097R0 — Contracts for C++: Support for Virtual Functions, we are interested in the proposed solution and encourage further work, independent of whether it is in P2900 or not.

SF	F	N	A	SA
23	11	3	5	2

Result: Consensus

EWG, St. Louis, 2024-06-28, Poll 2

P3097R0 — Contracts for C++: Support for Virtual Functions, we would like to see this paper merged into P2900 and progress contracts with virtual function support.

SF	F	N	A	SA
18	15	5	1	2

Result: Consensus

As a result of this polling, support for virtual functions was added to the Contracts proposal in revision [P2900R8].

However, in the lead-up to the February 2025 WG21 meeting in Hagenberg, concerns were raised in [P3506R0] and in [P3573R0] regarding the design that EWG had previously approved. The criticism focused mainly on the caller/callee model being novel and having insufficient deployment and usage experience in the field, being a departure from the classic model, and being complex and "known to be incomplete and [...] not known to be correct." [P3591R0], the rebuttal paper to [P3506R0] and [P3573R0], argued that the caller/callee model should be retained.

After discussion at the Hagenberg meeting, EWG came to the conclusion that the caller/callee model is insufficiently mature and too little understood for inclusion in C++26, that no proposal for another model is on the table, and that, therefore, support for `pre` and `post` on virtual functions should be removed from the Contracts proposal entirely.

EWG, Hagenberg, 2025-02-11, Poll

P2900: disallow `pre`/`post` contracts on virtual functions entirely.

SF	F	N	A	SA
20	24	13	14	2

Result: Consensus

This poll reversed EWG's previous decision in St. Louis. The feature was removed in revision [P2900R14].

3.3.3 Defaulted and Deleted Functions

Two reasons explain why it is ill-formed for functions defaulted on their first declaration to have function contract assertions. One reason is that such a function is thought to have the "default interface" generated by the compiler, and since function contract assertions are part of the interface

of a function, they cannot be part of that default interface. The other reason is that functions defaulted on their first declaration will frequently be trivial special member functions.

A call to such functions can be replaced by a no-op or a call to `std::memcpy`. Compilers perform such replacement as an optimization to remove the overhead of a function call, and many libraries perform this replacement explicitly. However, if a trivial constructor could have function contract assertions, they would be skipped if such replacement took place, even if contract checks were enabled and the user expected such checks to happen. This silent elision of contract checks is surprising and potentially unsafe. On the other hand, making a function defaulted on its first declaration implicitly *nontrivial* when the user adds a function contract assertion to it could cause performance regressions (even if the contract assertion was ignored) and changes to the compile-time semantics of the program (because triviality is detectable via traits) and thus could violate several design principles in [P2900R14]. We, therefore, do not allow such a function to have function contract assertions.

This problem was first considered in [P2932R3]. The paper offered the choice between (a) allowing contract assertions on trivial special member functions and allowing the checks to be elided or (b) allowing contract assertions on functions defaulted on their first declaration but rendering such functions nontrivial.

The November 2023 WG21 meeting in Kona clearly revealed that neither solution can gain consensus in SG21; instead, a compromise was adopted whereby adding a function contract assertion to a trivial function is ill-formed (allowing later evolution in either direction). Furthermore, the authors recognized that the same rule should apply to any function defaulted on its first declaration, since `= default` carries with it the notion of opting into a "default" interface generated by the compiler and function contract assertions are not part of such a default interface. This solution was adopted by poll, and the alternatives were rejected.

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, allow defaulted special member functions to have preconditions and postconditions without affecting the function's triviality, as proposed in P2932R2 Proposal 1.A.

SF	F	N	A	SA
0	5	3	3	6

Result: No consensus

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, make it ill-formed for a trivial defaulted special member function to have preconditions and postconditions, as proposed in P2932R2 Proposal 1.B.

SF	F	N	A	SA
1	7	4	5	1

Result: No consensus

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, make it ill-formed for any function defaulted on its first declaration to have preconditions and postconditions on that declaration, as proposed in P2932R2 Proposal 1.C.

SF	F	N	A	SA
12	4	2	1	0

Result: Consensus

Shortly after that meeting, SG21 realized that we had overlooked the case of deleted functions. Adding a function contract assertion to a deleted function is useless because such an assertion will never be checked (because the function will never be called); further, allowing function contract assertions on deleted functions but not on defaulted functions would be surprising and inconsistent. This case was, therefore, made ill-formed as well.

SG21, Teleconference, 2023-12-07, Poll

For the Contracts MVP, a deleted function having preconditions or postconditions is ill-formed.

SF	F	N	A	SA
10	4	3	0	0

Result: Consensus

3.3.4 Constructors and Destructors

The syntactic restriction that a nonstatic data member may not be named without an explicit `this->` in the predicate of a precondition assertion on a constructor or a postcondition assertion on a destructor has been introduced via [P2510R2] at the November 2024 WG21 meeting in Wrocław. The motivation and history for this syntactic restriction is discussed in that paper as well as in Section 3.6.2 below.

3.3.5 Await and Yield Expressions

During revision of the Contracts proposal ahead of forwarding it to EWG and LEWG for design review, Lewis Baker pointed out that the paper contained a design hole because it did not discuss what happens if a coroutine suspends during the evaluation of a contract assertion, which might happen if an *await-expression* or *yield-expression* appears as a subexpression of a contract predicate.

Three options were discussed.

1. Disallow a *await-expression* or *yield-expression* from appearing as a subexpression of a contract predicate.
2. Allow a predicate evaluation to be abandoned if a coroutine is being unwound while in the middle of evaluating a contract predicate.
3. Treat unwinding during a contract assertion as a contract violation. This situation, since it is a distinct form of detecting a violation, would warrant a new value for `detection_mode` because

violation handlers might want to act differently on this event. In particular, a violation handler might reasonably allow continuation from all such violations since a coroutine that is being abandoned might be considered acceptable.

Option 2 was discarded as subpar because cases occur in which such behavior would silence a bug that should have been detected, for many of the same reasons that an exception thrown during contract-predicate evaluation is detected and passed to the contract-violation handler. This left Options 1 and 3 as viable; Option 1 was chosen in accordance with [P2900R14]'s Design Principle 14, Choose Ill-Formed to Enable Flexible Evolution, because the entire picture for how contract assertions should interact with coroutines was not yet clear at that point.

This issue was discussed in an SG21 telecon during review of the Contracts paper. Option 1 was approved by poll; no separate paper for this issue was published.

SG21, Teleconference, 2024-02-29, Poll 1

For the Contracts MVP, make it ill-formed for `co_await` and `co_yield` to appear as a subexpression of a contract predicate.

SF	F	N	A	SA
6	12	0	0	0

Result: Consensus

Initially, the introduced rule was relevant only for `contract_assert` since an *await-expression* or *yield-expression* was already ruled out from appearing as a subexpression of the predicate of `pre` and `post` because a coroutine could not have function contract assertions on its declaration, only `contract_assert` inside its body. However, in revision [P2900R10], this restriction was lifted (see Section 3.5.2) so that the rule that an *await-expression* or *yield-expression* may not appear as a subexpression of the predicate is now relevant for all three kinds of contract assertions.

3.3.6 Pointers to Functions and Pointers To Member Functions

[P2900R14] supports the important use case that if a function is called through a function pointer, its function contract assertions are still being checked:

```
int f(int i)
  pre (i >= 0)
  post (r: r >= i);

int (*fptr)(int i) = f;
fptr(-3);  // contract violation
```

However, [P2900R14] does not allow function contract assertions to be placed on function pointers directly:

```
int (*fptr)(int i)
  pre (i >= 0)       // error
  post (r: r >= i);  // error
```

This restriction exists in the Contracts paper primarily because no proposal has been submitted to date that would specify how such placement of function contract assertions directly on function

pointers would work. [P3327R0] provides a thorough exploration of the design space for such a feature and points out all the challenges and questions that such a proposal would have to answer. In particular, one way or another, the function-contract-assertion sequence that applies to a function pointer would need to be stored *somewhere*: as part of the pointer type, as part of the pointer value, or as a property of the pointer variable declaration. However, each approach would have significant drawbacks.

The function-contract-assertion sequence being made part of the type system would mean that functions having different contracts would have different types. This modification would lead to code bloat: Adding or changing the function-contract-assertion sequence would trigger a new template instantiation in every case in which that function is used to deduce the template argument. In addition, this approach could lead to remote code breakage (code relying on functions having the same type), thus violating [P2900R14]'s Design Principle 15, No Client-Side Language Break, and raises questions about how contracts would interact with name mangling, type conversions, overload resolution, and so on.

If the function-contract-assertion sequence were made part of the runtime value of the pointer, run-time overhead in both space and time as well as other surprising consequences would inevitably result since we would essentially turn a static property into a dynamic property.

Finally, if the function-contract-assertion sequence were made a property of the pointer variable declaration (i.e., a form of "backchannel information" that is neither part of the pointer's type nor of its value — similar to `alignas`, certain attributes, and so on), then the problem arises that C++ does not have a mechanism for such backchannel information to propagate from typedefs, through expressions, across TU boundaries, or into templates, meaning that information about the contract assertions would be silently dropped in many cases — unless we standardize such a propagation mechanism as a new feature. Some implementation experience exists with vendor-specific attributes, but no actual proposal for such an approach is on the table, and developing such a proposal seems like a significant undertaking. This feature is, therefore, not part of [P2900R14].

A first exploration of placing contract assertions on function pointers — including many of the thoughts explored later in more depth in [P3327R0] — can be found in [N4110] in early Phase Three. The paper distinguishes between two separate questions.

1. What should happen when a function with function contract assertions is called through a function pointer? Should its assertions be checked?

2. Does it make sense to specify contract assertions on a function pointer directly?

Both questions are left open, but the paper notes that the second question does not seem to make sense since such a specification would effectively mean making the contract assertions part of the function type, which would raise lots of additional questions (e.g., how overloading on contracts would work and whether that would make any sense) and might cause many unwanted effects. The paper draws the conclusion that making these assertions part of the type system is not a viable solution.

Nevertheless, [P0246R0] and its companion paper, [P0247R0], introduced contract assertions on function pointers. The proposal makes those assertions apply *in addition* to any assertions on the function itself if that function is called through that pointer, but it does not make the assertions

part of the function pointer's type. The proposal notes that the assertions "should be bound to the object, not the type or value" of the pointer but does not clearly explain how such a design could be made to work.

The first concrete design for placing contract assertions on function pointers can be found in [N4415], [P0287R0], and [P0380R0]. These papers allowed contract assertions to be placed directly on function pointers and mandated that they must be the same (ODR identical) as the sequence of function contract assertions of the function assigned to that pointer and that restriction must be statically enforced. Assigning a function to such a pointer or assigning the value of a pointer to function to another such pointer was ill-formed if the restriction was violated. The contract assertions were neither a part of the type nor a part of the pointer value (Section 3.3.6). The problem of how to propagate contract information from `typedefs` was recognized but not solved in these papers: If a `typedef` is used to declare a function type or a pointer to function type, any contracts in the `typedef` declaration would not transfer to the function or to the pointer.

The motivation those papers give for this approach was to guarantee that when a function is called through a pointer, its function contract assertions are still checked. However, [P0380R1] found a much more effective way to satisfy this requirement by specifying that when a function is called through a pointer, its own function contract assertions are still evaluated as if it were called directly. At that point, the idea to place `pre` and `post` directly on function pointers was abandoned and made ill-formed. This notion carried over into the Contracts C++2a proposal [P0542R5] and ultimately into [P2900R14].

In [P3173R0], a major compiler vendor stated that a Contracts facility that fails to adequately support efficient use with pointer to functions is "woefully inadequate and unready for prime use." At the WG21 meeting in Tokyo (March 2023), where that paper was discussed, EWG took a poll on whether support for function pointers must be included in the Contracts proposal.

EWG, Tokyo, 2024-03-20, Poll 4

P2900R6 Contracts should specify contracts on function pointers in its Minimal Viable Proposal.

SF	F	N	A	SA
6	6	14	15	8

The poll showed significant interest in such a feature, although there was no suggestion how it could actually be made to work and no consensus that it *must* be included in the first version of a C++ Contracts facility (see [P3197R0]).

No concrete proposal to add `pre` and `post` to function pointers followed, but two papers proposed to carve out design space for adding such a feature as a later extension, anticipating scenarios in which a function-contract-assertion sequence could change the type of a function and interact with type deduction. [P3221R0] called for making it ill-formed to take the address of a function that has function contract assertions; SG21, however, decided that this suggestion would be too restrictive and unnecessary.

SG21, Teleconference, 2024-04-18, Poll 1

For the Contracts MVP, make it ill-formed to take the address of a function with a precondition or postcondition assertion, as proposed in P3221R0.

SF	F	N	A	SA
1	1	4	3	7

Result: Consensus against

[P3250R0] proposes a looser restriction than [P3221R0]: Rather than making it ill-formed to take the address of a function with function contract assertions, [P3250R0] merely makes it ill-formed to deduce the type of such a function. The address of a function with function contract assertions is treated as an overload set, making it ambiguous in a deduced context and selecting the version with no function contract assertions in case of a conversion to a type compatible with it. However, SG21 rejected this proposal as well.

SG21, St. Louis, 2024-06-26, Poll

In P2900, we want to carve out the address-of-function-with-contracts space for post-MVP contracts on function pointers as part of their type, as specified in P3250R0.

SF	F	N	A	SA
1	1	1	9	9

Result: Consensus against

The next proposal in this space was [P3271R1]. This paper does *not* propose to allow attaching function-contract specifiers to function pointers directly. Instead, it introduces so-called *function types with usage*, along with new syntax to define them:

```
int takes_and_returns_positive_numbers(int i) usage
  pre (i > 0)
  post (r: r > 0);
```

We can then declare a pointer to such a function type with usage, assign the address of a function to that pointer, and call the function through that pointer, which will perform a check of that usage type's contract around the call to the pointed-to function.

In this model, function usage types are a distinct kind of type different from function types. Two function usage types declared with different names are different types, even when they have the same contract-assertion sequence. When forming a pointer to a function usage type, the type of such a pointer is *similar* — using the core-language definition of that term[14] — to the type of a regular function pointer without a contract-assertion sequence, meaning that both types have the same value representation and are thus ABI compatible.

The approach in [P3271R1] combines some of the advantages of the type system approach, such as being compatible with templates and type deduction, with the advantages of the "property of the declaration" approach, such as no impact on ABI and no breakage of client code. However, the

[14] See [conv.qual]/2.

approach in [P3271R1] introduces a novel kind of entity to the language. This approach might or might not turn out to be a viable solution to the problem of function contract assertions on function pointers; SG21 decided not to consider this proposal for the C++26 timeframe but to revisit it as a possible future extension.

SG21, St. Louis, 2024-06-26, Poll

We want to spend more time considering P3271R0 "Function usage types", in the MVP timeframe.

SF	F	N	A	SA
0	1	9	8	2

Result: Consensus against

SG21, St. Louis, 2024-06-26, Poll

We want to spend more time considering P3271R0 "Function usage types", as a post-MVP extension.

SF	F	N	A	SA
12	8	2	0	0

Result: Consensus

Despite all these developments, at least one member of EWG still insisted that contracts on function pointers were a must-have requirement for the first version of a C++ Contracts facility. This insistence led to [P3327R0] being written, which addressed EWG and argued that Contracts should not be delayed while we figured out how to add contract assertions to function pointers and that placing contract assertions *directly* on function pointers (as opposed to some new entity, such as function usage types) seemed unlikely to be a viable solution. This paper was received well by EWG, which repolled whether support for function pointers must be included in the Contracts proposal. EWG showed strong consensus that we can move forward with the Contracts proposal without including this functionality.

EWG, Wrocław, 2024-11-18, Poll

p3327r0: contracts should specify contracts on function pointers in its Minimal Viable Proposal (P2900).

SF	F	N	A	SA
1	1	4	10	20

Result: Consensus against

This strong consensus in EWG unblocked the Contracts proposal, which could now proceed toward standardization without support for contract assertions on function pointers.

The latest proposal in this space is [P3583R0], yet another approach to introduce a new syntax to add contract assertions to function pointers. This paper was received with interest by SG21,

but various issues with the proposal were identified and the author was told to come back with a revision.

SG21, Hagenberg, 2025-01-13

We want to spend more SG21 time considering the design direction of P3583R0 "Contracts, Types & Functions" for contracts on function pointers, and encourage the author to come back with a revision.

SF	F	N	A	SA
1	5	4	2	0

Result: Consensus

In summary, despite a significant number of papers written on the matter, exploring at least five distinct approaches for how contract assertions could be added to function pointers, we are still, at the time of writing, in the exploratory stage; significant hurdles remain before an extension to add contract assertions to function pointers could be considered for the C++ Standard.

3.3.7 Function Type Aliases

Adding contract assertions to a function type alias is ill-formed in the Contracts proposal for essentially the same reason that adding contract assertions to function pointers is ill-formed: No proposal to add such a feature has been brought forward so far, and any such proposal would have to answer the question whether the function-contract-assertion sequence would become part of the type or should instead be treated as "backchannel information," with both approaches having significant limitations. (See discussion in Section 3.3.6 above and more detailed discussion in [P3327R0] and references therein.)

3.3.8 Use of C Variadic Functions Parameters

During its design review of [P2900R6], EWG pointed out that we must consider how C variadic function parameters should interact with contract predicates. This topic was subsequently considered in [P3119R1]. The paper came to the conclusion that, since any use of `va_start` within a contract predicate would need to be matched by a use of `va_end` in the same predicate but that such a requirement cannot be statically checked and thus would have to be made undefined behavior if violated, the better and safer option is to make using `va_start` inside a contract predicate ill-formed. This fix was adopted for the Contracts proposal.

SG21, Teleconference, 2024-04-04, Poll 8

For the Contracts MVP, make it ill-formed, no diagnostic required for a contract predicate to enclose a use of `va_start`.

SF	F	N	A	SA
5	10	1	0	0

Result: Consensus

3.4 Semantics

3.4.1 Name Lookup and Access Control

The rules for name lookup and access control inside the predicate of `contract_assert` are straightforward: Name lookup and access control occur as if the predicate expression were located in an expression statement at the location of the assertion statement. No other rule seems to make sense for `contract_assert`.

For `pre` and `post`, the question of how name lookup and access control should work is somewhat less straightforward since they are part of a function interface.

Initially the rule was that name lookup in contract assertions should happen as if the predicate expression appeared in a `decltype` in a trailing return type. This rule first introduced in [P2388R0] (more detailed motivation in [P2521R5]; earlier proposals did not explicitly discuss name lookup in `pre` and `post`). The wording in [P2388R4] took a slightly different path in which name lookup would happen as if the predicate expression appeared as the operand of a `noexcept` specifier on the function declaration. Upon closer consideration, both versions seem to lead to the same result.

Starting from [P2900R0], to match the behavior of an expression that is also evaluated, the predicate was instead treated as if it were the first expression in the body of the function, with the parameter names from the declaration that has the precondition or postcondition specifier.

The question of access control — in particular, whether `pre` and `post` on a member function declaration should have access to the *private* members of the class — goes back a bit further.

C++2a Contracts initially introduced a rule that access control is dependent on the access specifier of the member function itself. Precondition and postcondition specifiers on a public member function could reference only public members of the class. On a protected member function, they could additionally reference protected members, and on a private member function, they could additionally reference private members. These rules were motivated in [N4415] by a wish to prevent possible "abstraction leakage" through `pre` and `post`.

However, this rule was later discovered to significantly limit the use cases for precondition and postcondition assertions without a clear benefit. Bjarne Stroustrup expressed the need for a contract assertion on a member function to have private access to the implementation of the function's class so that the contract assertion could write efficient preconditions without having to expose extra functions in a public API that were not directly relevant to clients. This discussion led to a paper, [P1289R1], which contains a thorough discussion of this design aspect. The paper proposed to remove the restrictions on access control and allow that every member be used in precondition and postcondition specifiers on any member function. The paper was approved by EWG and adopted into the C++20 Working Draft.

EWG, San Diego, 2018-11-07, Poll

P1289 Access control in contract conditions: Proposal P1289R0 as presented

SF	F	N	A	SA
15	14	0	1	0

Result: Consensus

The allowance to access private members from `pre` and `post` was retained in Phase Four, as indicated in Section 4.3 of [P2521R4]. A more recent writeup of the motivation for this decision can be found in [P2388R4].

Name lookup for `pre` and `post` on a lambda expression deserves a special mention. When SG21 allowed `pre` and `post` on lambda expressions (see Section 3.2.1), they also approved the name lookup rules for this case as proposed in [P2890R1]. However, the name lookup rules proposed in that paper were later found to be defective, and no special rule for the lambda case was necessary at all. This discovery is explained in more detail in the successor paper [P2890R2]. The fix was approved by SG21.

SG21, Teleconference, 2023-12-14, Poll 1

For the Contracts MVP, adopt the name lookup rules for `pre` and `post` on lambdas proposed in P2890R2. Name lookup for `pre` and `post` on a lambda should follow the existing rules for `pre` and `post` on functions in the Contracts MVP: it should be performed as-if the predicate were at the beginning of the lambda's body.

SF	F	N	A	SA
10	5	0	0	0

Result: Consensus

3.4.2 Implicit `const`-ness

The Contracts proposal is built around a central purpose for contract assertions. Each contract assertion describes a single, discrete algorithm that identifies whether a contract violation has occurred. Importantly, these checks encode into a program parts of the plain-language contract that itself defines when the evaluation of that program is correct. For contract assertions to benignly provide information about the program to which they are being applied, rather than just producing a different program with functionally different behavior, they must never themselves alter the correctness of that program ([P2900R14]'s Design Principle 1, Prime Directive). Contract predicates whose presence or evaluation has side effects violating that principle are said to be *destructive* (an idea that was first introduced in [P2712R0]).

The Contracts proposal strives to discourage such destructive predicates ([P2900R14]'s Design Principle 6, No Destructive Side Effects). However, we cannot, in general, reject predicates with side effects in the core-language sense, and even if we could, side effects in the core-language sense are not necessarily destructive side effects, and destructive side effects are not necessarily side effects in the core-language sense. (See Section 3.5.8 for further discussion.)

That said, the most frequent reason a contract predicate is destructive is a direct change to the state of an object that is relevant to the function's behavior, such as a function parameter or other variable visible outside the cone of evaluation of the contract predicate itself. Invoking a function that is semantically nonmutating — even if not strictly free of side effects in the core-language sense — is often an indication that the predicate itself is not destructive. C++, luckily, already has a concept for describing when operations are expected to be semantically nonmodifying operations: `const`.

Therefore, a reasonable approach to reducing the chance that contract predicates are inadvertently destructive is to minimize their ability to execute non-`const` operations outside their cone of evaluation. Following this approach, [P2900R14] implicitly adds `const` to any *id-expression* naming a variable declared outside of the contract predicate itself, in the same way as, for example, a `const` member function implicitly adds `const` to any *id-expression* naming a member variable of the class; this approach has since informally been dubbed `const`-*ification*. This approach does not (and cannot) prevent *all* destructive side effects (no approach that could achieve this is possible), but it removes an entire class of common bugs by making contract assertions such as the following ill-formed, without breaking correctly written assertions.

In the experience of those who have supported contract-assertion frameworks at scale, the two primary misuses of a Contracts facility are significantly diminished by `const`-ification.

1. Often, users introduced to a contract-checking facility will decide that it is a quick way to validate the return value of a function. Many users also tend to prefer terse code that compacts as much as possible into each individual expression. When these choices combine, many developers leap to the following solution for validating the results of a function:

   ```
   contract_assert(widget.do_something() == true);   // error: cannot call non−const
                                                     // member function here
   ```

   Such misuse of assertions quickly results in problems in an environment where development is done with assertions enabled and production releases turn them off; suddenly, the production version of the program just does nothing, which certainly fails to be the intended behavior of the program.

2. Another, subtler problem arises when users decide that the contract assertion is responsible for not only detecting a bug, but also fixing it:

   ```
   void f(std::vector<int> v)
     pre( std::is_sorted(v.begin(), v.end())
          ? true
          : std::sort(v.begin(), v.end()), true );
   ```

   Here, a user has decided, when the passed-in list of items is not sorted, to just sort it. This mental model of the responsibilities of contract-assertion predicates again falls apart when an unchecked build is deployed and the function enters untested territory where it must deal with an unsorted input.

While `const`-ification is an effective approach at catching destructive predicates at compile time, it has two significant tradeoffs.

The first tradeoff is that non-`const`-correct APIs will not compile when used in a contract predicate:

```
struct Widget {
  bool is_valid();     // nonmodifying but missing const
};

void f(Widget& w)
  pre(w.is_valid());  // error: cannot call non−const member function here
```

To avoid the compile error above, we would have to add the missing `const` to the API of `Widget` or if that API cannot be modified, work around it with a `const_cast`:

```
void f(Widget& w)
  pre(const_cast<Widget&>(w).is_valid());  // OK
```

Notably, while the *id-expression* naming the variable inside the contract predicate is `const`-ified, the original underlying variable is still non-`const` (unless declared as such by the user) and thus may be modified, including in a contract predicate, without causing undefined behavior. To avoid the suboptimal ergonomics of naked `const_cast`s, the following convenience macro can be used to access the underlying, non-`const`-ified variable inside a contract predicate:

```
#define UNCONST(x) const_cast<std::add_lvalue_reference_t<decltype(x)>>(x)

void f(Widget& w)
  pre(UNCONST(w).is_valid());  // OK
```

Importantly, this macro will *not* remove the `const` from a variable that actually has `const` on its declaration, significantly reducing the risk of misusing this form of `const_cast`.[15] In the future, we might add more convenient escape hatches for the above; several such escape hatches are discussed in Section 4.4 of [P3261R1].

The second tradeoff is that `const`-ification affects overload resolution and, therefore, leads to inconsistent semantics for expressions inside and outside contract predicates with respect to overload resolution:

```
class Widget {
  bool f()       { return false; }
  bool f() const { return true; }
};

void f(Widget w)
  pre(w.f())              // false
{
  assert(w.f());          // true
  contract_assert(w.f()); // false
  if (w.f()) /* ... */;   // true
}
```

[15] This macro will, however, remove `const` from a member variable name in a `const` member function, and for the same reason, it can remove the `const` from a by-value capture in a (non-`mutable`) lambda when the captured variable is not `const`. Only a built-in operator would be able to fix these issues.

However, to be effective at catching destructive predicates without being too limiting in what predicates can be expressed, `const`-ification *has* to affect overload resolution. In particular, `const`-ification is the only approach to the destructive predicates problem proposed so far that enables the first contract assertion to compile and rejects the second contract assertion in the following example:

```
void f(std::vector<int> v)
  pre(std::is_sorted(v.begin(), v.end());     // OK, const overload selected

void f(std::vector<int> v)
  pre((std::sort(v.begin(), v.end()), true); // error: no const overload available
```

Arguably, an overload set giving semantically different results for the `const` and the non-`const` overload is itself a bug and not something that our Contracts feature should strive to support, or at least, supporting such cases is less important than preventing a large number of bugs due to destructive predicates. Various examples of such broken overload sets have been given in [P3478R0] as well as informally on the SG21 reflector, but none have been found so far that are used in contract assertions in any real-world code.

A thorough analysis of `const`-ification, its advantages and downsides, and possible alternative approaches is available in [P3261R1]; case studies of the effect `const`-ification has on real-world code can be found in [P3268R0] and [P3336R0]; and `const`-ification is also explicitly mentioned in "Implementers Report," [P3460R0]. The most up-to-date rationale of having `const`-ification in [P2900R14] can be found in [P3591R0], which in turn is a response to concerns about `const`-ification that are published in [P3506R0] and in [P3573R0].

Like many aspects of [P2900R14], `const`-ification (both the mechanism and the set of entities to which it applies) has a storied history that goes back over two decades. The first incarnation of `const`-ification appears in the very first Contracts proposal, [N1613], in Phase One. The [N1613] Contracts proposal prevented modifications of parameters and local entities by making them implicitly `const` in the contract assertion; [N1773] clarified that this protection from modification is achieved by specifying that the contract scope behaves as if it were a `const` member function.

Phase Two proposals also did not have such a provision because they were based on macro assertions which, like the existing C `assert` macro, are not themselves a core-language construct and, therefore, do not have the capability to add implicit `const`-ification to entities referenced in the contract predicate.

Phase Three proposals also did not feature `const`-ification. [N4110], [N4415], and other papers suggested that modifications should not be allowed but proposed no mechanism for preventing or discouraging them. The first draft for C++20 Contracts wording, [P0542R0], contained no provision for this restriction. Revision [P0542R4] made any modification of *any* object in a contract predicate, as well as any other side effects, undefined behavior. This proposal was approved by EWG.

In Phase Four, the undefined behavior was removed again since the new proposal (unlike C++2a Contracts) was designed to prevent adding new undefined behavior to the language. [P2388R0] extensively discussed the possibility of making parameters implicitly `const` in the contract predicate and concluded that such a possibility would be nice but would lead to a discrepancy between expressions inside and outside a contract predicate; with `const`-ification, overload resolution would

select different overloads inside and outside, which was deemed too confusing. The paper discussed other options as well, such as making implicit copies. [P2388R1] added more rationale about why `const`-ification was undesirable: Some functions do not modify parameters but do not mark them `const` (for example, in legacy APIs), and these functions need to be callable in contract predicates.

As work on the Contracts proposal progressed, SG21's opinion on this question shifted. Discouraging accidental modifications in contract predicates was deemed important since such changes are a significant source of bugs. Paper [P3071R0] proposed making variables of automatic storage duration, `*this`, and the result type in postcondition implicitly `const` in contract predicates, making modifications of these objects in a contract predicate ill-formed without an explicit `const_cast`. [P3071R0] did not `const`-ify variables of static storage duration,[16] did not attempt to apply `const` to pointers (one of the reasons being that doing so would make raw pointers inconsistent with smart pointers or other user-defined types with an overloaded `operator->`), and did not attempt to invent a form of *deep* `const`.[17]

A revision, [P3071R1], added `const` treatment for lambda captures of `this`. This revision was approved by SG21 with no votes against and is now part of the specification.

SG21, Teleconference, 2023-12-14, Poll 2

For the Contracts MVP, adopt D3071R1 "Protection against modifications in contracts" as presented.

SF	F	N	A	SA
6	10	3	0	0

Result: Consensus

The tradeoffs of `const`-ification (incompatibility with non-`const`-correct APIs, inconsistent semantics of expressions inside and outside contract predicates with respect to overload resolution) were discussed at the time, but the advantages of preventing (at least some) bugs due to accidental modifications were deemed to be more significant by SG21, in particular because workarounds for `const`-ification are available.

However, concerns about `const`-ification in contract assertions were brought up again during the design review of [P2900R6] in EWG, which was divided on the matter.

[16] Originally, the more pragmatic route seemed to be to limit `const`-ification to only those variables that exist solely for the evaluation of the function currently being evaluated. As more users were exposed to the idea and more experiments were conducted, we clearly saw that the cognitive load of special rules regarding what is and is not `const`-ified was significant, and a simpler rule of `const`-ifying all variables declared outside the contract-assertion predicate was much easier to understand.

[17] [P3261R2] offers a thorough exploration of the possibility of adding deep-`const`, but enabling it would require extensive changes to the language. More importantly, without a user-controllable customizable way to propagate deep-`const`, the inconsistencies between user-defined and built-in types would grow unboundedly.

EWG, St. Louis, 2024-06-24, Poll

P2900R7: local variables, `*this`, and the return value in contract predicates should not be implicitly `const`.

SF	F	N	A	SA
9	5	4	10	5

Result: No consensus for change but a strong divide

The matter was thus referred back to SG21. The discussion there pointed out that, in addition to the known tradeoffs, `const`-ification would make migrating from C `assert` and similar macro-based facilities to Contracts more difficult. To aid such migration, [P3257R0] proposed to undo `const`-ification just for `contract_assert`, and [P3281R0] proposed to undo `const`-ification for all kinds of contract assertions; neither proposal gained consensus in SG21.

SG21, Teleconference, 2024-05-16, Poll 1

For the Contracts MVP, remove the rule that in the predicate of an assertion statement (`contract_assert`), variables with automatic storage duration are implicitly `const`, as proposed in P3257R0 Proposal 1.

SF	F	N	A	SA
8	1	2	5	4

Result: No consensus

SG21, Teleconference, 2024-05-16, Poll 2

For the Contracts MVP, remove the rule that in the predicate of a contract assertion of any kind (`pre`, `post`, and `contract_assert`), variables with automatic storage duration and the result object of a function are implicitly `const`, as proposed in P3281R0.

SF	F	N	A	SA
8	4	0	4	3

Result: No consensus

Case studies on the impact of `const`-ification in real-world code were published in [P3268R0] and in [P3336R0]; none found the negative impact to be significant. Both papers concluded that insufficient motivation exists to remove `const`-ification from the Contracts proposal. [P3270R0] offered a principled-design analysis of the `const`-ification question.

However, concerns over `const`-ification and, in particular, its effect on overload resolution were brought up *again* in the lead-up to the EWG review of [P2900R10] at the WG21 meeting in Wrocław. An initial round of discussion about these renewed concerns showed a divide in SG21 on the question of `const`-ification.

SG21, Teleconference, 2024-10-03, Poll

We favour removing constification from P2900R8, pending further discussion.

SF	F	N	A	SA
5	7	2	6	3

These concerns were subsequently published in [P3478R0]. As a response to these concerns, [P3261R1] contained the most thorough analysis of `const`-ification yet. It lists all known concerns and all known alternative approaches and discusses, in great detail, the tradeoffs of each approach with respect to these concerns. After the discussion of that paper in SG21, the question was polled again. The poll result re-established that SG21 has strong consensus to retain `const`-ification.

SG21, Teleconference, 2024-10-10, Poll 1

Remove constification from P2900R8, as proposed in P3261R1 Proposal 1.

SF	F	N	A	SA
3	1	2	10	8

Result: Consensus against

This discussion also led to the adoption of a design improvement proposed in [P3261R1]. In particular, SG21 recognized that applying `const`-ification to only local variables renders the feature harder to understand. In addition, [P3261R1] showed concerns related to common (grossly incorrect) workarounds that were suggested for problems with `const`-ification, such as marking automatic variables `static` or moving them to namespace scope. SG21 thus decided to apply `const`-ification consistently to all variables declared outside the contract-assertion predicate, including those with static storage duration.

SG21, Teleconference, 2024-10-10, Poll 2

Apply constification to all variables, as proposed in P3261R1 Proposal C (including referenced from within a lambda-expression within a predicate).

SF	F	N	A	SA
6	14	2	0	3

Result: Consensus

The poll immediately above represents the status quo in [P2900R14]. Nevertheless, some EWG members still had concerns that `const`-ification is an inconsistent or inappropriate solution to the problem of rejecting certain destructive predicates at compile time. [P3478R0] argued that `const`-ification should be removed from the Contracts proposal; an updated revision of the more thorough analysis of `const`-ification, [P3261R2], argued that the design should stay as it is and published data showing that several large codebases could be migrated to using `contract_assert` with `const`-ification to implement their assertion macros and that no major problems caused by `const`-ification were encountered. In fact, these experiments revealed several bugs in existing assertions. Both papers

were discussed side by side in EWG at the WG21 meeting in November 2024 in Wrocław. The resulting EWG poll showed consensus for retaining `const`-ification.

EWG, Wrocław, 2024-11-19, Poll

P3261R1 / P3478R0: P2900 shall not have const-ification by default.

SF	F	N	A	SA
10	4	9	19	12

Result: Consensus against

This decision by EWG notwithstanding, the opponents of `const`-ification continued to publish papers arguing against it, most recently as part of a longer list of concerns, in [P3506R0] and [P3573R0], about the Contracts design. The rebuttal paper, [P3591R0], argued *again* why `const`-ification is the correct design for [P2900R14]. The Contracts proposal had already been forwarded to CWG and LWG, at that point, for wording review and inclusion into the C++26 Working Paper, yet these papers spawned another discussion of `const`-ification in EWG at the February 2025 WG21 meeting in Hagenberg. The result of that discussion was even stronger consensus against removing `const`-ification.

EWG, Hagenberg, 2025-02-11, Poll

P2900: remove constification.

SF	F	N	A	SA
9	7	6	37	14

Result: Consensus against

3.4.3 The Result Binding

A significant number of postconditions assert conditions on the return value of a function; as such, the ability to refer to this return value in the predicate of a postcondition assertion is an important requirement for a C++ Contracts feature. Postcondition assertions offer a syntax to introduce a user-defined identifier that denotes the return object of a function *directly* and allows its use in the predicate expression. This is a novel feature not available with assertion macros.[18]

The syntax to refer to the return object, as well as the exact semantics of the feature, varied throughout the history of Contracts for C++.

In Phase One, the first Contracts proposal, [N1613], used the `return` keyword to refer to the return value in an `out { ... }` block. In [N1669], this syntax was changed, and users were given the ability to define their own name for the return value via the syntax `postcondition (r) { ... }` where `r` is the introduced name.

Phase Two proposals did not support postcondition assertions and, therefore, did not have a feature for referring to the return value of a function.

[18] Assertion macros can emulate this feature to some extent in case the function body *names* the object being returned; however, such emulation is impossible without subtly changing program semantics when the function returns a prvalue.

In Phase Three, [N4110] explored several syntactic options: using a keyword (as in [N1613]), having a syntax for introducing a user-defined result name (as in [N1669]), or using the name of the function itself, i.e., the novel idea proposed in [N4110]. [P0246R0] proposed to use the `return` keyword, e.g., `[[post : return > 0 ]]`. [N4293] proposed to use the name of the function, e.g., `[[post: f > 0]]` where `f` is the name of the function to which this postcondition assertion appertains. [P0287R0] removed the ability to refer to the return value entirely "to keep the proposal minimal." Finally, after EWG's adoption of [P1344R1], [P0380R0] introduced the familiar Phase Three syntax of a user-defined name that immediately precedes the colon, e.g., `[[ensures r : r > 0]]` or `[[post r : r > 0]]`.

This last syntax was initially retained in Phase Four. When SG21 lost consensus on the attribute-like syntax, [P2737R0] again revived the idea — as part of the "condition-centric syntax" proposal — to use a keyword instead of a user-chosen identifier, proposed `result` as a new reserved identifier for this purpose, and provided some motivation for this idea. However, the idea of a reserved identifier was rejected by SG21 (see electronic poll results in [P2885R3]). Finally, the current syntax with a user-defined identifier followed by a colon (called the *result name introducer* in [P2900R14]) immediately before the predicate expression was added via [P2961R2] (the "natural syntax" proposal).

The question of the syntax for naming the return object was raised again in [P3249R0] and [P3210R0], which observed that the syntax was inconsistent with the pattern-matching syntax proposed in [P2688R3] and posited that syntactic consistency between the two features would be desirable.

[P3249R0] proposed to directly change the syntax `post (r: ...)` that introduces a name `r` for the return object, where `r` is any user-defined identifier, to the syntax `post (let r => ...)`, which follows the [P2688R3] syntax for introducing a name in a pattern match `let r => ...`. However, because at the time Contracts appeared likely and pattern matching seemed unlikely to ship in C++26 and because the pattern-matching syntax might still change since the token `=>` is also claimed by the implication operator proposal ([P2971R2]), [P3249R0] was abandoned.

[P3210R0] acknowledged this uncertainty over the timing and shape of the pattern-matching proposal by instead proposing a two-staged approach: first, to adopt the [P2737R0] syntax for postcondition assertions (which had been rejected by electronic poll; see Section 3.2) for the Contracts proposal, i.e., to replace the *result-name-introducer* with a predefined identifier, `result`, to refer to the result object in the postcondition predicate; second, to consider a future extension to that syntax, to ship in the same version of C++ as [P2688R3], of the form `post (`α`)` where α has the same syntax as α in the [P2688R3] syntax `result match { `α`; }`. The fundamental idea of [P3210R0] was that a postcondition assertion can be made an implicit pattern-match body against the return value of the function, i.e., that "a postcondition *is* a pattern match." However, this idea was not received favorably by SG21. (Note that we can already use a pattern match inside a postcondition assertion with [P2900R14].) The first proposal of [P3210R0] was rejected by SG21 both for the predefined identifier `result` and for any other predefined identifier or token.

SG21, Teleconference, 2024-09-05, Poll 1

For the Contracts MVP, adopt Proposal 1 of P3210R0, replacing the syntax `post(r : r > 0)` with the P2737 syntax `post(result > 0)`, where `result` is a predefined name referring to the result object.

SF	F	N	A	SA
2	0	2	3	7

Result: Consensus against

SG21, Teleconference, 2024-09-05, Poll 2

For the Contracts MVP, consider a change that replaces the syntax `post(r : r > 0)` with the syntax `post(@ > 0)` where `@` is some other token referring to the result object.

SF	F	N	A	SA
1	2	3	3	5

Result: Consensus against

The second proposal in [P3210R0] was postponed by SG21 to a possible future extension.

SG21, Teleconference, 2024-09-05, Poll 3

We want to spend more time considering Proposal 2 of P3210R0, adding an extension to the postcondition syntax that matches the syntax of the pattern matching proposal P2688, as a post-MVP extension.

SF	F	N	A	SA
1	6	2	3	0

Result: Consensus

Poll 3 motivated the author to come back to SG21 with a revision, [P3210R2], that proposed a slightly different path toward unifying postcondition-assertion syntax and pattern-match syntax. After discussion of this paper, SG21 decided that we have consensus against any such syntax changes and further do not have consensus to pursue any such syntax unification going further.

SG21, Teleconference, 2024-10-17, Poll 1

In P2900, replace the syntax `post(r : r > 0)` for the *result-name-introducer* in a postcondition assertion with the syntax `post(let r => r > 0)` as proposed by P3210R2 Proposal 1.

SF	F	N	A	SA
1	1	6	5	4

Result: Consensus against

SG21, Teleconference, 2024-10-17, Poll 2

We are interested in implicitly making a postcondition assertion a pattern match body against the return value of the function as proposed by P3210R2 Proposal 2.

SF	F	N	A	SA
1	0	5	5	6

Result: Consensus against

SG21, Teleconference, 2024-10-17, Poll 3

We recommend to EWG that they consider the relationship between postcondition assertion syntax and pattern matching syntax.

SF	F	N	A	SA
2	7	3	2	3

Result: No consensus

No further proposals for changing the syntax for the result name introducer in a postcondition assertion were submitted after this decision.

In addition to the syntax, we also need to specify the semantics of the introduced name. Is it a reference to the result object of the function, a copy of it, or something else? What exact type and value category does it have?

[N1669] proposed that the introduced name should be a `const` reference to the result object. The C++2a Contracts wording provided only a vague specification, saying that the name "represents the glvalue result or the prvalue result object of the function." Finally, a full specification for the result binding[19] was proposed in [P3007R0]: It should be an lvalue naming the return object of a function, similar to how the identifiers introduced by a structured binding are names, not references; `decltype(r)` should be the return type of the function; and several other rules cover certain corner cases. The paper contains an extensive motivation and exploration of the available design space. This specification was approved by SG21.

SG21, Teleconference, 2023-12-14, Poll 3

For the Contracts MVP, adopt P3007R0 "Return object semantics in postcondition specifiers" with the return-name being a const lvalue referring to the return object.

SF	F	N	A	SA
6	10	2	0	0

Result: Consensus

In addition to the above, [P2900R14] contains several other rules regarding the result binding that originated in earlier papers.

[19] At the time, the terminology for the result binding kept fluctuating: It was initially called the "return name," then the "result name," before finally settling on "result binding" in [P2900R14].

The rules for *nontemplated* functions with a deduced (`auto`) return type were inherited from [P1323R0]: When the declared return type of a nontemplated function contains a placeholder type, the result name introducer shall be present only in a definition, and types or names dependent on the deduced return type are *not* treated as dependent types or names, so they do not require `template` and `typename` disambiguators. These rules were approved by EWG in Phase Three.

EWG, San Diego, 2018-11-07, Poll

P1323R0 Contract postconditions and return type deduction: Disallow naming the return value in a postcondition if the function has a deduced return type.

SF	F	N	A	SA
3	5	7	8	0

Result: No consensus

EWG, San Diego, 2018-11-07, Poll

P1323R0 Contract postconditions and return type deduction: Treat the name of the return value as having dependent type for template entities, and allow such naming for templated functions. For non-templated functions, allow such naming without assumption of dependence and only for definitions.

SF	F	N	A	SA
1	13	2	4	0

Result: Consensus

Following the above EWG decision, the wording of [P1323R0] was approved by CWG. However, in Phase Four and during development of the Contracts paper, the wording regarding dependent types and names got moved around and transformed and, as a result, became less clear, leading to a question about the design intent in "Implementers Report," [P3460R0], regarding whether such types and names should be treated as dependent or implemented by delayed parsing (with a recommendation to do the latter).

This question was clarified in Section 1.1 of [P3483R1], which reconfirms the original design intention of [P1323R0] and contains a thorough discussion of the issue complete with code examples. This reconfirmation of the design approved by EWG was approved by SG21.

SG21, Teleconference, 2024-10-31, Poll

Apply to P2900 the changes proposed in P3483R0, except those proposed in Section 1.3.

SF	F	N	A	SA
5	8	1	0	0

Result: Consensus

The known implementation strategy for this rule is to store the postcondition assertion as "token soup" until the return type is known and to properly parse it afterward (also known as *delayed*

parsing and already necessary in other parts of the C++ language, such as parsing inline definitions of member functions).

Another rule, namely that it is ill-formed to introduce a result name if the function return type is `void`, originates from the proposed wording in [P2388R4] and was carried over to [P2900R14].

3.4.4 Function Parameters in Postconditions

If a function parameter is odr-used by a postcondition assertion predicate, that function parameter must have reference type or be `const`. The basic motivation is given in [P2900R14]: If a parameter of a function is referred to in a postcondition predicate, we must be able to reason about the value of this parameter in that predicate irrespective of the definition of the function, which is not possible and leads to very surprising results if that definition is allowed to modify the value. While this issue has been known since Phase Three at the latest, the exact way in which it was addressed varied throughout history.

Throughout Phases One through Three, no explicit `const` was required on a parameter declaration if that parameter was odr-used in a postcondition predicate. In Phase One, starting with [N1613], such parameters were considered to be *implicitly* `const` in the contract predicate. Subsequent papers suggested that, if a parameter is odr-used in `post`, it should be ill-formed to modify the value of that parameter in the function body by making the parameter implicitly `const`, but those proposals did not yet offer a concrete set of rules for how to accomplish that goal.

Phases Two and Three did not explicitly consider this case until [P0542R4], which proposed that, if a parameter is odr-used in a postcondition predicate, modifying its value in the function body is undefined behavior; this version was eventually merged into the C++20 Working Paper via [P0542R5].

In [P2388R1], a nonreference function parameter referenced in a postcondition must be declared `const` on the function definition (but not necessarily on the function declaration). In [P2388R2], this rule was changed to a requirement that such a parameter must be declared `const` on every declaration of the function.

SG21, Teleconference, 2022-03-10, Poll

Adopt a change based on option #2 from the paper P2521R1 section 4.3: if the postcondition uses a non-reference parameter: require it to be `const` objects.

SF	F	N	A	SA
5	6	0	1	0

Result: Consensus

Section 3.13 of [P2521R5] contains detailed rationale for this change, including an explanation of why the problem does not occur in other programming languages.

[P2829R0] proposed to revert this rule to `const` being required only on the definition to support the `const`-on-definition programming style; however, this paper was rejected by SG21.

SG21, Teleconference, 2023-07-13, Poll 1

Adopt P2829R0 as presented into the Contracts MVP.

SF	F	N	A	SA
0	1	0	6	14

Result: Consensus against

[P2466R0] contains detailed rationale regarding why the rule does not apply to reference parameters. Essentially, reference parameters do not introduce new objects but are merely an alternative way to refer to existing objects in the caller's scope, and as such, the underlying object could be modified by the function anyway, even if the reference is `const`.

The case of a parameter having a dependent type and the question whether the `const` must be explicit on the template declaration or can be part of that dependent type were first considered in Section 1.3 of [P3483R0]. The paper proposed that, as a clarification of the existing design, the `const` may be part of a dependent type. However, discussion of that paper in SG21 quickly recognized that such a rule is not merely a clarification, but actually an open design question and has more implications and tradeoffs than stated in the paper. As a result, the topic was separated into a distinct paper, [P3489R0], which contains a more thorough discussion of the tradeoffs of both options. SG21 discussed this separate paper and initially chose the option that the `const` must be explicit.

SG21, Teleconference, 2024-11-07, Poll 9

For P2900, require every parameter declaration to have an explicit `const` qualifier, as proposed in P3489R0 Option D1.

SF	F	N	A	SA
2	11	3	1	2

Result: Consensus

SG21, Teleconference, 2024-11-07, Poll 10

For P2900, allow the `const` on a parameter declaration to be part of a dependent type, as proposed in P3489R0 Option D2.

SF	F	N	A	SA
2	5	7	1	2

Result: Consensus but weaker than Poll 9

However, wording review of [P2900R11] in CWG raised the concern that this rule also excludes parameters declared via a type alias of a `const` type, which is unfortunate. For example, the following code should be well-formed:

```
using const_int_t = const int;
void f(const_int_t i) post (i > 0);
```

Examples are also available in which the parameter is declared via a type alias that is itself dependent. This case should also be well-formed:

```
template <typename T>
void f(std::add_const_t<T> t) post(t > 0);
```

Thus, Section 3 of [P3520R0] (which offers a more thorough analysis) proposed to undo SG21's previous decision and instead adopt the rule that the `const` on a parameter declaration can be part of a dependent type. This rule was approved by both SG21 and EWG.

SG21, Wrocław, 2024-11-22, Poll 3

Remove the requirement that the `const` qualifier of a non-reference parameter odr-used in a `post` has to be explicitly spelled on the parameter declaration, as proposed in P3520R0 Section 3.

SF	F	N	A	SA
5	8	1	0	0

Result: Consensus

EWG, Wrocław, 2024-11-22, Poll

P3520r0 contracts, EWG supports the paper's suggested fix for item 3, which mandates that the type of a parameter must be `const` under certain conditions as outlined in the paper.

SF	F	N	A	SA
14	19	4	0	0

Result: Consensus

Further, when these issues were discussed, support for `pre` and `post` on virtual functions was still part of the proposal. (Such support was removed later; see Section 3.3.2.) For `post` on a virtual function, [P3484R2] recognized the problem that `const` could also be dropped on an overriding function and the parameter modified in the body of that override, even if it is declared `const` in all declarations of the overridden function that is being called. The paper provided a thorough analysis of this problem space and described six different solutions to the problem and their tradeoffs. Two solutions were determined to be unviable: one because it would add new undefined behavior to the language, which is actively user hostile and goes against [P2900R14]'s Design Principle 13, Explicitly Define All New Behavior, and the other because it appears to be unspecifiable or nonimplementable. The remaining four solutions are viable, but all have significant tradeoffs. SG21 thoroughly discussed these solutions and their tradeoffs, as described in [P3484R2], and then polled, which led to the adoption of a new rule in revision [P2900R11] that a parameter odr-used in the predicate of a postcondition assertion must also be declared `const` on all declarations of all overrides of that function.

SG21, Teleconference, 2024-11-14, Poll 1

In P2900, disallow odr-using any non-reference parameter in `post` on a virtual function, regardless of whether that parameter is declared `const`, unless that function is marked `final` or is a member function of a class marked `final`, as proposed in P3484R2 Option V1.

SF	F	N	A	SA
0	2	3	9	2

Result: Consensus against

SG21, Teleconference, 2024-11-14, Poll 2

In P2900, require that if a non-reference parameter is odr-used in `post` on a virtual function, that parameter must also be declared `const` in every declaration of every overriding function, as proposed in P3484R2 Option V2.

SF	F	N	A	SA
7	8	0	1	0

Result: Consensus

SG21, Teleconference, 2024-11-14, Poll 3

In P2900, require that if a non-reference parameter is odr-used in `post` on a virtual function, that parameter must also be declared `const` in every definition of every overriding function, as proposed in P3484R2 Option V3.

SF	F	N	A	SA
2	6	3	5	0

Result: No consensus

SG21, Teleconference, 2024-11-14, Poll 4

In P2900, add a non-normative recommendation that a diagnostic be issued when an overriding function omits `const` from the declaration of a non-reference parameter odr-used in the postcondition of an overridden function as proposed in P3484R2 option V5.

SF	F	N	A	SA
0	2	3	9	2

Result: Consensus against

However, this point became moot later when support for `pre` and `post` on virtual functions was removed from the proposal entirely in revision [P2900R14].

Adding support for postcondition assertions on coroutines (see Section 3.5.2) required determining how to deal with nonreference parameters odr-used in such postconditions, given that a coroutine will initialize copies of the parameters in the coroutine frame with modifiable xvalues referring to the

original parameters, ignoring any `const` qualification on those original parameters,[20] which means that such parameters may be moved from.

The first proposal to add contract assertions to coroutines, [P2957R0], stated that parameter names in function contract assertions should refer to the original parameter objects, rather than to their copies, but did not go into detail about whether or how the problem of moved-from parameters should be addressed. The next revision of that proposal, [P2957R1], recognized this problem and proposed to disallow postcondition assertions on coroutines to avoid the problem; the paper also mentions the possible alternative of allowing only postcondition assertions that do not odr-use function parameters.

[P3387R0] provides a thorough analysis of this problem and discusses eight possible solutions.

1. Do not allow `pre` or `post` on coroutines at all.
2. Allow `pre` on coroutines but not `post`.
3. Allow `post` on coroutines, but do not allow odr-using nonreference parameters in its predicate.
4. Allow odr-using nonreference parameters in the predicate of `post`; an *id-expression* naming a nonreference parameter refers to the copy made for the coroutine state.
5. Allow odr-using nonreference parameters in the predicate of `post`; an *id-expression* naming a nonreference parameter refers to the original object, and
   (a) the parameter copy made in the ramp function is copy-constructed instead of move-constructed
   (b) the *id-expression* is ill-formed if the parameter type has a nontrivial move constructor (i.e., the parameter can have a moved-from value)
   (c) no further provision is added, i.e., the *id-expression* may refer to a moved-from value, even if the parameter is declared `const` by the user

The paper discusses the tradeoffs of all options, explains why options 4 and 5 are nonviable, and proposes option 3. A subsequent revision of the original proposal, [P2957R2], proposes option 3 and was approved by SG21 and EWG. (See Section 3.5.2 for the poll results and further discussion.)

Finally, the rule that odr-using an array parameter by a postcondition-assertion predicate is ill-formed was adopted by SG21 via Proposal 1 of [P3119R0].

SG21, Teleconference, 2024-04-04, Poll 7

For the Contracts MVP, make it ill-formed to odr-use an array parameter from the predicate of a postcondition, as specified in Proposal 1 of P3119R0.

SF	F	N	A	SA
3	12	0	0	0

Result: Consensus

[P3119R1] contains a detailed motivation for this rule, complete with code examples.

[20] See [dcl.fct.def.coroutine]/13.

3.4.5 Objects Passed and Returned in Registers

When the exact semantics for the return object in a postcondition assertion were developed in [P3007R0] and then adopted into the Contracts proposal, the interaction between objects *returned* via registers and postcondition assertions was known, and the appropriate dispensation was added to the proposal along with the rest of the semantics (see Section 3.4.3).

However, the interaction between objects *passed* via registers and precondition and postcondition assertions was recognized, via [P3487R0], only much later, i.e., after a bug in the GCC implementation of caller-side checking of caller-facing postconditions was observed. The Contracts proposal lacked a special dispensation for this case, and as a result, caller-side checking of preconditions and postconditions was made impossible without an ABI break (because they were not allowed to observe the final parameter object since that might exist only within the function body when a type is eligible to be passed via registers). On the other hand, introducing such a dispensation means creating cases like the `RandomInteger` example in [P2900R14], where the presence of `mutable` subobjects in a parameter passed via registers can lead to spurious failures of an otherwise correctly written postcondition.

[P3487R0] provides an in-depth analysis of the different options and their tradeoffs, which were discussed and polled in SG21. The option chosen was Option R7 from [P3487R0], adding the dispensation for precondition and postcondition assertions to observe the original parameter object and choosing to refrain from introducing any special rule to avoid cases like the `RandomInteger` example because of the unfavorable tradeoffs of any such rule.

SG21, Teleconference, 2024-11-07, Poll 1

For P2900, remove postconditions, as proposed in P3487R0 Option R1.

SF	F	N	A	SA
0	0	3	9	7

Result: Consensus against

SG21, Teleconference, 2024-11-07, Poll 2

For P2900, make it ill-formed to odr-use any function parameter in a postcondition, as proposed in P3487R0 Option R2.
Result: Unanimous consent to not consider this option

SG21, Teleconference, 2024-11-07, Poll 3

For P2900, make it ill-formed to odr-use any non-reference function parameter in a postcondition, as proposed in P3487R0 Option R3.

SF	F	N	A	SA
0	5	3	7	3

Result: No consensus

SG21, Teleconference, 2024-11-07, Poll 4

For P2900, allow the postcondition to see the caller-side version of a parameter; make it ill-formed to odr-use a non-reference function parameter in a postcondition unless the parameter is of scalar type, as proposed in P3487R0 Option R4.
Result: Unanimous consent to not consider this option

SG21, Teleconference, 2024-11-07, Poll 5

For P2900, allow the postcondition to see the caller-side version of a parameter; make it ill-formed to odr-use a non-reference function parameter in a postcondition if the type of the parameter satisfies the requirements for being passed via registers, unless it is of scalar type, as proposed in P3487R0 Option R5.
Result: Unanimous consent to not consider this option

SG21, Teleconference, 2024-11-07, Poll 6

For P2900, allow the postcondition to see the caller-side version of a parameter; make it ill-formed to odr-use a non-reference function parameter in a postcondition if the type of the parameter satisfies the requirements for being passed via registers and has at least one `mutable` subobject, as proposed in P3487R0 Option R6.

SF	F	N	A	SA
0	0	3	9	7

Result: Consensus against

SG21, Teleconference, 2024-11-07, Poll 7

For P2900, allow the postcondition to see the caller-side version of a parameter; do not introduce any restrictions to odr-use a non-reference function parameter in a postcondition, as proposed in P3487R0 Option R7.

SF	F	N	A	SA
11	4	1	1	0

Result: Consensus

SG21, Teleconference, 2024-11-07, Poll 8

For P2900, clarify that the postcondition is not allowed to see the caller-side version of a parameter, as proposed in P3487R0 Option R8.
Result: Unanimous consent to not consider this option

The implementers report, [P3460R0], requested clarification on whether, if the result object is passed in registers and referred to in postcondition predicates, the same materialized temporary must be used in each such postcondition predicate. This question was discussed in detail in Section 1.2 of [P3483R1]. The paper proposed a clarification that the compiler is allowed to make extra trivial copies of the return object, and the postcondition assertions may then refer to those copies. Those

copies, however, need to be performed in sequence with the evaluation of the postcondition assertions; a code example illustrating this situation can be found in that paper as well as in [P2900R14]. That clarification was approved by SG21 alongside the other clarifications proposed in [P3483R1]. (See Section 3.4.3 for the poll result.)

3.4.6 Not Part of the Immediate Context

Detailed rationale for why precondition and postcondition assertions are not part of the immediate context and a discussion of design alternatives are provided in [P2388R4]. Essentially, silently removing functions from an overload set via SFINAE because they have precondition or postcondition specifiers with an invalid predicate expression is undesirable, and doing so would violate the Contracts Prime Directive (Design Principle 1 of [P2900R14]); silently discarding the precondition and postcondition assertions themselves in such cases is equally undesirable.

Note that the rationale in [P2388R4] dates from a time when name lookup in precondition and postcondition predicates was modeled as if the predicate were the expression inside a `noexcept` specifier (see Section 3.4.1), and the "not part of the immediate context" rule (as well as the wording for it) was equally modeled after the wording for `noexcept` specifiers. Nevertheless, the rationale for the rule still applies equally to [P2900R14] and fits well with the design principle that has since been established regarding noninterference of contract assertions with the surrounding program.

3.4.7 Function Template Specializations

The rule that function contract assertions of an explicit specialization of a function template are independent of the function contract assertions of the primary template originates from the C++2a Contracts wording ([P0542R5]) and is required for consistency with the design principle that function contract assertions on a function must check a subset of the plain-language contract of *that* function ([P2900R14]'s Design Principle 10, Contract Assertions Check a Plain-Language Contract).

3.4.8 No Implicit Lambda Captures

[P2834R1] revealed that a contract assertion that is part of a lambda can trigger an implicit lambda capture, thus violating the Contracts Prime Directive (Design Principle 1 of [P2900R14]). In particular, odr-using an entity in a contract assertion that is not otherwise odr-used in that lambda will cause that entity to be implicitly captured as part of the generated closure object, which can affect the size and/or alignment of the closure type as well as whether the closure type is trivially copyable and/or a standard-layout class.

Although the C++ Standard specifically calls out those properties as something that the user should not rely on, the mere presence of a contract assertion can nevertheless alter the observable semantics of a program, including compile-time effects such as changing the layout of a class just by adding a contract assertion to the code, and [P2900R14] is otherwise carefully trying to avoid these kinds of changes:

```
constexpr auto f(int i) {
  return sizeof( [=] pre (i > 0) {} ); // captures i by value
}
```

```
struct X {
  char data[f(0)];  // sizeof(X) == 4, or 1 with the pre above removed.
};
```

[P2834R1] and its successor paper, [P2932R2], provide various other code examples demonstrating the issue, including one in which adding a contract assertion to the program leads to an expensive copy and another where it changes overload resolution.

The proposed solution in these papers was that, within the predicate of a contract assertion that is attached to a lambda (`pre` and `post`) or within the body of a lambda (`contract_assert`), the odr-use of a local entity should *not* implicitly capture that entity, although those references would still attempt to reference the captured variable and the program would thus be ill-formed if nothing else within the lambda body caused the capture to happen.

[P2890R2] provided further analysis of the issue and concluded that just not performing the capture would not work; normative wording would be needed to make sure that the name referenced the capture and did not attempt to denote the local variable in an enclosing scope:

```
static int i = 0;
void test() {
  int i = 1;
  auto f = [=] { contract_assert(i > 0); }; // Which i is referenced here?
}
```

Instead, [P2890R2] proposed that triggering a capture from a contract predicate should make the program ill-formed or, alternatively, that the entire issue should be ignored or relegated to a compiler warning. A later revision of the other paper, [P2932R3], clarified the mechanism of the restriction and was later adopted.

All these options were considered by SG21, and the proposal to make the program ill-formed was adopted.

SG21, Teleconference, 2023-12-07, Poll 4

For the Contracts MVP, if an entity implicitly captured by a lambda expression L is only referenced within the preconditions and postconditions of L and within contract assertions inside the body of L, the program is ill-formed.

SF	F	N	A	SA
7	7	1	3	0

Result: Consensus

3.5 Evaluation and Contract-Violation Handling

3.5.1 Point of Evaluation

All Contracts for C++ proposals to date agree that precondition assertions should be evaluated when a function is called, postcondition assertions should be evaluated when a function returns, and assertion statements (if such a thing is included in the proposal) should be evaluated when control flow reaches that statement. Those seem to be the only sensible choices. For example, the

C++2a proposal [P0542R5] states that "a precondition is checked ... immediately before starting evaluation of the function body" and "a postcondition is checked immediately before returning control to the caller of the function."

[P2900R14] makes this specification more precise by specifying exactly how those evaluations are sequenced with respect to other operations that happen when a function is called and when it returns.

In particular, precondition assertions must be evaluated after the initialization of function parameters; otherwise, odr-using these parameters in a precondition predicate would be impossible. Symmetrically, postcondition assertions must be evaluated prior to the destruction of function parameters, because otherwise odr-using these parameters in a postcondition predicate would be impossible. Note that for an ABI that performs destruction of function parameters on the callee side (notably, the Microsoft ABI), this means that many postconditions must be checked on the callee side.

Further, since the result binding in a postcondition assertion refers *directly* to the result object of a function (at least, if its type is not trivially copyable; see Section 3.4.3 for an explanation of this rule), postcondition assertions must be evaluated after this result object has been initialized. This evaluation order is directly observable:

```
X f(X* ptr)
  post(r: &r == ptr) {
  return X{};
}

int main() {
  X x = f(&x);
}
```

Here, if `X` is not trivially copyable, the postcondition assertion is *guaranteed* to pass for the call from `main` below, because by the time it is evaluated, `x` in `main` has already been initialized, and the result binding in the postcondition assertion refers directly to that object.

Finally, postcondition assertions must be evaluated after the destruction of local variables since the evaluation of the destructors of such variables is conceptually part of executing the function body and might affect whether the postconditions of the function are satisfied.

Note that postconditions are not checked when control leaves a function other than by returning normally. In particular, postconditions are not checked when a function exits via an exception. Writing contract assertions that are checked in such circumstances is not directly possible with [P2900R14] but will become possible with future extensions, such as procedural function interfaces (see [P0265R0] and [P2755R1]).

With the point of evaluation for all kinds of contract assertions clearly defined, we can now understand an important difference between precondition and postcondition *assertions* on one hand and (plain-language) *preconditions* and *postconditions* on the other. Precondition assertions, postcondition assertions, and assertion statements fundamentally work in the same way and are distinguished from one another only by syntax and by their points of evaluation, and (plain-language) preconditions and postconditions are distinguished by who — the caller or the callee — is responsible for ensuring that they are true.

In most — but importantly, not in all — cases, precondition and postcondition assertions are used to check preconditions and postconditions, respectively. In some cases, to check a (plain-language) precondition, we might use, at the beginning of a function body, an assertion statement (e.g., to insulate the check from the caller if it is considered to be an implementation detail) or even a postcondition assertion (e.g., because the precondition predicate can be evaluated algorithmically more efficiently after having evaluated the function body first).

3.5.2 Coroutines

Revision [P2900R10] of the Contracts paper added support for function contract assertions on coroutines. Section 3.5.4 of [P2900R14] describes the motivation for the proposed design. For the motivation and history of the nonreference parameter rule for coroutines, please see Section 3.4.4.

Since coroutines were not part of the C++ Standard until C++20, support for contract assertions on coroutines was not considered until Phase Four. [P2900R14] allows `contract_assert` inside the definition of a coroutine since `contract_assert` at that location has the same behavior as a `contract_assert` inside the definition of any other function.

[P2957R0] first recognized that the fundamental design principle of Coroutines in C++ — the coroutine-ness of a function is an implementation detail — means that precondition and postcondition assertions on a coroutine declaration must apply not to the coroutine itself but to the *ramp function* that creates it. The paper proposed to add support for precondition and postcondition assertions on coroutines based on that principle.

However, perhaps due to the somewhat unintuitive nature of coroutines in C++ and to Coroutines being still a relatively recently standardized feature at the time, the proposal was not well understood and thus also not well received by SG21. A counterproposal, Proposal 5 of [P2932R3], argued that too many open questions lingered about how Contracts should work on coroutines and that the only reasonable solution, therefore, was to withhold support for precondition and postcondition specifiers on coroutines and to come back to this question after shipping an initial version of the C++ Contracts facility.

Both papers were discussed by SG21 at the November 2023 meeting in Kona. During this discussion, SG21 recognized that nonreference parameters that are odr-used in postcondition predicates are a problem for Coroutines (see Section 3.4.4) and did not have a satisfactory solution at the time. As a result, [P2957R0] was rejected, and function contract assertions on coroutines were made ill-formed.

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, allow preconditions and postconditions on coroutines, with the semantics proposed in P2957R0.

SF	F	N	A	SA
3	1	2	7	6

Result: No consensus

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, allow preconditions on coroutines, with the semantics proposed in P2957R0; make it ill-formed for a coroutine to have postconditions.

SF	F	N	A	SA
1	3	5	7	2

Result: No consensus

SG21, Kona, 2023-11-10, Poll

For the Contracts MVP, make it ill-formed for a coroutine to have preconditions and postconditions, as proposed in P2932R2 Proposal 5.

SF	F	N	A	SA
6	8	1	2	3

Result: Consensus

However, in [P3173R0], a major compiler vendor stated that "a Contracts facility that fails to adequately support efficient use with coroutines ... is woefully inadequate and unready for prime use." When [P3173R0] was discussed at the WG21 meeting in Tokyo (March 2023), EWG took a poll on whether coroutine support must be included in the first version of the C++ Contracts facility that we ship.

EWG, Tokyo, 2024-03-20, Poll 5

P2900R6 Contracts should specify contracts on coroutines in its Minimal Viable Proposal.

SF	F	N	A	SA
6	3	11	19	7

While the above poll did not indicate consensus one way or the other, it did indicate a somewhat split room. Therefore, while SG21 initially did not plan to include support for Coroutines in the initial Contracts proposal (see [P3197R0]), the feature was prioritized again to increase consensus on the proposal as a whole.

This work resulted in papers [P3387R0] and [P2957R2]. The solution proposed in these papers was approved by SG21 and is part of [P2900R14] today.

SG21, Teleconference, 2024-09-26, Poll

Forward the design for contracts on coroutines proposed in D2957R2, as presented, to EWG for design review.

SF	F	N	A	SA
8	8	1	1	0

Result: Consensus

This proposal was then also approved by EWG.

EWG, Wrocław, 2024-11-18, Poll

P2900r11 shall contain the changes proposed in P2957R2, adding support for coroutines to the design of contracts.

SF	F	N	A	SA
16	16	2	1	0

Result: Consensus

A third paper on the topic, [P3251R0], also supported the solution proposed in [P3387R0] and [P2957R2]. In addition, it provided some discussion of how precondition and postcondition assertions that apply to the values returned from within the coroutine could be specified as precondition and postcondition assertions on the *promise type* of the coroutine. The paper was discussed by SG21; the paper did not propose to add anything new, so no poll was needed.

3.5.3 Observable Checkpoints

The decision to base the Contracts proposal on top of the observable checkpoints proposal [P1494R5] was motivated by the concerns raised in [P2680R1], [P3173R0], and [P3285R0] that the evaluation of a contract assertion could be affected by undefined behavior.

We found that distinguishing between two separate kinds of situations is useful:

1. Contract checks performed with the *observe* semantic being optimized out due to undefined behavior in subsequent code
2. Contract checks performed with *any* evaluation semantic (including *enforce* and *quick-enforce*) being optimized out or otherwise affected due to undefined behavior that occurs while evaluating the contract predicate of *that* check

We found that the first kind of situation can be removed entirely by making contract assertions observable checkpoints (see code examples in [P2900R14]). The second kind, however, is of a different nature and is not directly helped by the introduction of observable checkpoints; this situation is discussed further in Section 3.6.1.

[P3328R0] provided the necessary wording to add observable checkpoints to the Contracts proposal to address the first kind of undefined behavior issue (as described above). This paper was approved by SG21 at the same St. Louis meeting, establishing a dependency between the two proposals (which were both under development at the time).

SG21, St. Louis, 2024-06-26, Poll

In P2900, make the beginning of evaluation of a contract predicate when evaluating a contract assertion an observable checkpoint, as proposed in P3328R0, conditional on P1494R3 being adopted into the WD.

SF	F	N	A	SA
12	8	1	0	0

Result: Consensus

SG21, St. Louis, 2024-06-26, Poll

In P2900, make the contract-violation handler returning normally when it is invoked from a contract assertion having the observe semantic an observable checkpoint, as proposed in P3328R0, conditional on P1494R3 being adopted into the WD.

SF	F	N	A	SA
12	8	1	0	0

Result: Consensus

At the February 2025 WG21 meeting in Hagenberg, the observable checkpoints proposal, [P1494R5], was adopted into the C++26 Working Paper alongside [P2900R14], thus resolving the dependency.

3.5.4 Evaluation Semantics: *ignore*, *observe*, *enforce*, *quick-enforce*

[P2900R14] offers four possibilities for what should happen when a contract assertion is evaluated at run time.

- *ignore*: Evaluation of the contract assertion has no effect; the predicate expression, however, is still parsed and odr-used.
- *observe*: The predicate is checked; if a contract violation occurs, the contract-violation handler is called, and if the contract-violation handler returns normally, control flow continues as normal.
- *enforce*: The predicate is checked; if a contract violation occurs, the contract-violation handler is called, and if the contract-violation handler returns normally, the program is terminated in an implementation-defined way.
- *quick-enforce*: The predicate is checked; if a contract violation occurs, the program is *immediately* terminated in an implementation-defined way.

Unlike earlier Contracts proposals, where these possibilities had to be derived from the interplay of various switches, such as the assertion level, build mode, and continuation mode, [P2900R14] makes these options first-class citizens called *evaluation semantics* that are explicitly named. After the experience of C++2a Contracts, we found that such explicit enumeration of the available semantics helps users compartmentalize and reason about the possible behaviors of a contract assertion in a way that they would otherwise not. This thoughtful consideration in turn promotes better understanding and more effective usage of the proposed Contracts feature.

Each proposed evaluation semantic exists to serve an important use case.

- *ignore*: As a runtime semantic, *ignore* can be used in situations where the cost of runtime checking (runtime overhead, code size, and so on) is too high to be worth the benefit of increased confidence in the code's correctness. This strategy can be appropriate in situations of high confidence that no contract violation will occur and/or the possible negative consequences of a contract violation occurring are sufficiently limited. As a compile-time semantic (see Section 3.5.12), *ignore* can be used if we cannot afford the increased compile times incurred by evaluating contract predicates in `consteval` and `constexpr` functions during constant evaluation.
- *observe*: As a runtime semantic, *observe* can be used in situations where the user wants to be informed of contract violations without interrupting the flow of the program. In many situations, this strategy is not advisable because continuing past a contract violation is likely to execute incorrect code and cause undefined behavior. The use case for *observe* is the introduction of new contract assertions to a mature legacy codebase that is *known* to be stable in production (i.e., any undefined behavior it exhibits is most likely "benign"), allowing the user to find additional defects in that codebase without affecting the production system. As a compile-time semantic, *observe* can be used to inform the user of compile-time contract violations (via compiler warnings) without breaking the build (making the program ill-formed) in cases where the cost of such breakage would be unacceptable.
- *enforce*: As a runtime semantic, *enforce* can be used in situations where the user does not wish to continue program execution past a contract violation to thus avoid executing incorrect code and wants to be informed of the contract violation and its location via the default contract-violation handler or via their own installed contract-violation handler that provides custom logging, triggers a breakpoint, and so on. As a compile-time semantic, *enforce* is the correct choice if the user wants to ensure that code containing contract violations during constant evaluation does not enter production.
- *quick-enforce*: An alternative to *enforce*, *quick-enforce* can be used in situations where the overhead in code size caused by the contract-violation handling machinery is unacceptably high and/or the attack surface of the program after a contract violation has occurred must be minimized. Such use cases include binary hardening and safety-critical applications.

More detailed motivation for having both the *observe* and the *enforce* semantic can be found in [P2877R0], and more detailed motivation for having the *quick-enforce* semantic separate from *enforce* can be found in [P3191R0].

For the terminating semantics *enforce* and *quick-enforce*, the mode of termination is implementation-defined to be one of the following.

- `std::terminate` is called.
- `std::abort` is called.
- Execution is terminated immediately.

This implementation freedom allows compilers to cater for different use cases. For example, `std::abort` or `std::terminate` could be useful if the user wishes to install a `SIGABRT` handler or

a `std::terminate_handler`, respectively, but on some platforms, those tools might be unavailable and/or running any user-defined code on contract violation might be undesirable. In particular, this implementation freedom allows *quick-enforce* to be implemented by immediately terminating the program with a single instruction, such as Clang's `__builtin_trap` or `__builtin_verbose_trap`, if the contract check did not succeed. This option avoids calling any library functions, performing any exception handling, and so on to minimize the available attack surface. With `__builtin_verbose_trap`, diagnostic information can still be obtained by attaching a debugger or via a symbolicated crash log. This use case is described in more detail in [P3191R0].

Importantly, an implementation is not tied to one choice for any evaluations of contract assertions in a given program. For example, it is entirely reasonable that the *enforce* semantic will call `std::abort` (to be backward compatible with the `assert` macro), while the *quick-enforce* will execute `__builtin_trap` when the predicate evaluates to `false` and call `std::terminate` when evaluation of the predicate exits via an exception, all within the same program. To implement the *enforce* semantic, the implementation has to surround any potentially throwing predicate evaluation with a `try`/`catch` block (see Section 3.5); on the other hand, to implement the *quick-enforce* semantic, that overhead can be avoided by evaluating the predicate evaluation in an implicit `noexcept` context that immediately terminates the program by calling `std::terminate` when predicate evaluation exits via an exception.

Note that the three allowed termination modes all perform *erroneous* termination and return a nonzero value to the host environment. Termination modes in C++ that can be employed to perform a *nonerroneous* termination, such as `std::exit`, `std::quick_exit`, and `std::_Exit`, are not allowed since a contract violation is *always* a defect in the program code. An overview of all possible termination modes in C++ and their differences and tradeoffs as well as a detailed rationale for the choice of termination modes proposed here can be found in Section 1 of [P3520R0].

Note further that for the termination modes that call into a C or C++ Standard-Library function (`std::abort` or `std::terminate`), an implementation is free to perform the actions equivalent to such a call without actually performing the call and thus without having to link the Standard Library. This permission makes deploying terminating evaluation semantics possible in a wider range of scenarios.

Beyond the four proposed semantics, [P2900R14] explicitly allows implementations to add their own custom evaluation semantics as extensions. This flexibility is useful both for integrating existing tooling with the proposed Contracts feature and for exploring possible future extensions of [P2900R14]. For example, while the *assume* semantic (see Section 2.3) is purposefully not included in the first version of the Contracts feature that we propose to ship, a compiler can provide this evaluation semantic as a vendor-specific extension, and thus we can gain usage experience with it in the field.

The proposed set of four evaluation semantics is the result of over two decades of evolution, both within WG21 and outside of it. For example, usage experience with the `BSLS_REVIEW` and `BSLS_ASSERT` macros in Bloomberg's BDE libraries was instrumental for the inclusion of both the *observe* and *enforce* semantics — often within the same translation unit or even the same function.

Focusing on the work within WG21 during the last two decades, we note that the set of options for *what happens* when a contract assertion is evaluated (which we now call evaluation semantics)

has undergone many permutations, both in functionality and in terminology — perhaps more than any other aspect of this proposal. In the remainder of this section, we summarize the evolution of the proposal with regard to which evaluation semantics are available; in the following section, we will discuss the motivation and evolution of the mechanisms through which a particular evaluation semantic is selected.

The very first Contracts for C++ proposal, [N1613], offered the same two options that the C `assert` macro offers and that many other programming languages offer: A contract assertion is either *checked* or *ignored*. However, in the next revision, [N1669], the behavior was changed to "always checked" and remained so throughout Phase One. On contract failure, "a customizable callback" (see Section 3.5.9) should be called that "defaults to `std::terminate`" — an early precursor to today's *observe* and *enforce* semantics.

[N3604] reintroduced the option to *ignore* a contract instead of always checking it. As the paper notes, such an unchecked semantic is essential because it sets contract assertions apart from existing control flow constructs ([P2900R14]'s Design Principle 12, Contract Assertions Are Not Control Flow). Every Contracts for C++ proposal since [N3604] contains such an unchecked semantic.

[P0166R0] provides an enumeration of possible "effects on contract violation" as follows: "fail-fast and terminate," "handle and continue," and "throw and recover." The possibility of providing the *assume* semantic (again, without yet using that terminology) was discussed in [P0147R0].

[P0246R0] provides an *ignore* and an *observe* semantic and notes that implementations "may have a build mode" where returning from the violation handler terminates (*enforce* semantic) or results in undefined behavior (note that this particular manifestation of undefined behavior is distinct from a contract violation with the *assume* semantic because here the contract is checked).

The Contracts C++2a proposal, [P0542R5], specified that an unchecked contract assertion that would not evaluate to `true` is undefined behavior. [P0542R5] thus implicitly replaced *ignore* with *assume* as the offered unchecked semantic. Opinions differ on whether this design change was intentional or the result of a wording defect that crept in as [P0542R5] was making its way through the different WG21 subgroups and into the C++20 Working Paper.

After [P0542R5] was merged into the C++20 Working Paper, EWG realized that offering the *assume* semantic by default greatly increases the probability that a contract assertion might inadvertently introduce undefined behavior to the program. [P1290R3] and [P1321R0] proposed to remove this ability, i.e., to effectively change the unchecked semantic back from *assume* to *ignore*. [P1710R0], [P1711R0], [P1730R0], and [P1786R0] proposed to add a global "assumption mode" that would offer a choice between *assume* and *ignore*; none of these papers, however, were adopted.

EWG, San Diego, 2018-11-07, Poll

P1321: UB in contract violations — Axioms are only assumed in `default` and `audit` build modes:

SF	F	N	A	SA
3	3	15	4	2

EWG, San Diego, 2018-11-07, Poll

P1321: UB in contract violations — Remove the introduction of assumptions by the presence of non-axiom contracts:

SF	F	N	A	SA
15	5	3	3	7

EWG, Kona, 2019-02-19, Poll

P1290: Avoiding undefined behavior in contracts

SF	F	N	A	SA
5	5	10	7	7

The question of whether to include the *assume* semantic eventually became moot as the Contracts feature was removed from the C++20 Working Paper entirely. When it became clear that Contracts were not going to be part of Standard C++ any time soon, but the possibility to inject an assumption into the program was still a useful and needed feature, EWG and SG21 agreed to pursue this feature separately from the Contracts facility.

EWG, Prague, 2020-02-13, Poll

We want assumptions now and independent of future contract facilities

SF	F	N	A	SA
18	5	1	3	3

EWG, Prague, 2020-02-13, Poll

We like the proposed semantics for assumptions

SF	F	N	A	SA
18	5	4	2	0

EWG, Prague, 2020-02-13, Poll

We want exploration on a mode which can check assumptions, including side effects

SF	F	N	A	SA
1	0	9	9	5

SG21, Prague, 2020-02-14, Poll

Assumptions should proceed independently of contracts.

SF	F	N	A	SA
9	8	5	6	5

SG21, Prague, 2020-02-14, Poll

We should discourage the ability to turn assertions into assumptions in contract proposals.

SF	F	N	A	SA
14	4	2	3	5

SG21, Prague, 2020-02-14, Poll

The contract facility should provide a facility to turn assertions into assumptions.

SF	F	N	A	SA
7	4	2	3	10

SG21, Prague, 2020-02-14, Poll

The contract facility should provide a kind of assertion that is never assumed.

SF	F	N	A	SA
17	4	3	0	2

SG21, Prague, 2020-02-14, Poll

The contract facility should provide a kind of assertion that can be assumed.

SF	F	N	A	SA
6	3	4	8	5

SG21, Prague, 2020-02-14, Poll

Contract design should provide a facility to turn assumptions into assertions.

SF	F	N	A	SA
17	9	0	0	1

This standalone assumption feature was proposed by [P1774R8] in the form of the `[[assume]]` attribute, which was included in C++23. (An in-depth discussion of the relationship between

assertions and assumptions can be found in [P2064R0].) SG21 subsequently decided to omit *assume* from the Contracts proposal.

SG21, Teleconference, 2020-10-06, Poll

We agree to progress towards contract checking to enforce software safety first, and enable assumptions of injected facts at a later time.

SF	F	N	A	SA
7	4	2	0	1

Result: Consensus

One problem until then had been that the different options for the behavior of evaluating a contract assertion were not actually explicitly enumerated; rather, they were discussed in the context of various properties that such an evaluation could have — whether it was checked or unchecked, which "build mode," "continuation mode," or "assumption mode" was being used, and so on. A major step forward was the introduction of the term *semantic* for this behavior and the idea to name and enumerate all the possible semantics. Stemming from a summary originally sent to the reflector by Hyman Rosen, this idea was formally introduced in [P1332R0], which listed the following five possible semantics: *ignore*, *assume*, *check_never_continue*, *check_maybe_continue*, and *check_always_continue*. In this terminology, *check_never_continue* maps to [P2900R14]'s *enforce* and *check_maybe_continue* maps to [P2900R14]'s *observe*. More discussion of this set of five possible semantics can be found in [P1333R0] and [P1334R0].

The idea of the *check_always_continue* semantic — in an attempt to introduce a semantic that would minimize the impact that a checked contract assertion would have on optimizations — was that the compiler could always assume that the contract-violation handler would return normally. However, the *check_always_continue* semantic was later found to be perhaps difficult or even impossible to implement and without much practical benefit. While constructing examples where *check_always_continue* could potentially have a benefit is possible, all such examples ultimately transform broken code into different broken code, and in no known case would this semantic correct broken code. In [P1429R3], the *check_always_continue* semantic was, therefore, dropped.

In an attempt to increase the consensus on C++2a Contracts, [P1607R1] proposed to minimize the proposal to retain only *ignore* and *check_never_continue*, relegating *assume* and *check_always_continue* to "additional building blocks."

In Phase Four, after the C++2a Contracts proposal had been removed from the C++20 Working Paper and C++20 shipped without Contracts, renewed discussions took place about which semantics should be offered in a new Contracts proposal.

[P2076R0] summarized previous differences of opinion about Contracts and noted the disagreements that had occurred about whether a "continuation facility" (i.e., *observe*) should exist at all; for example, [P1711R0] had noted that continuation was "a major complicating factor and most unusual in contract systems outside the C++ proposal."

[P2182R1] attempted to define a minimum viable feature set and proposed to remove continuation (i.e., *observe*) from that feature set and return to an "enforced or ignored" model. This model was the one proposed in the first iteration of the Contracts proposal in Phase Four, [P2388R0]

(which provided these two evaluation semantics via the two proposed build modes, *No_eval* and *Eval_and_abort*).

Bjarne Stroustrup's much-discussed paper, [P2698R0], highlighted the programs and use cases for which unconditional termination on contract failure was not an option. Since continuing into buggy code after a contract violation is also usually not an option, his suggested solution was to instead throw an exception via an *Eval_and_throw* semantic to thus attempt to recover from a contract violation further up the call stack.

At the February 2023 WG21 meeting in Issaquah, the problem highlighted by [P2698R0] was deemed serious enough that the SG21 roadmap was altered to address this problem sooner and to relegate agreement on a syntax to a later point in time.

SG21, Issaquah, 2023-02-08, Poll 4

In the SG21 roadmap (P2695R0), swap the target dates for producing a design for syntax and violation handling.

SF	F	N	A	SA
11	3	5	0	0

Result: Consensus

The possibility of not terminating upon contract violation was further explored in [P2784R0], which lists several use cases and proposes an alternative approach to throwing (see also Section 3.6.6), a design featuring so-called *abortable components*. Another alternative to termination was explored in [P2852R0], which proposed to effectively replace the *enforce* semantic with *observe*, and let the user choose whether to terminate the program on contract violation by terminating in a user-provided contract-violation handler.

These different ideas culminated in [P2877R0], which proposed to adopt a set of three semantics: *ignore*, *observe*, and *enforce*, covering all known use cases. Note that *observe* is not actually necessary to avoid termination, and the same effect can be achieved by throwing from the contract-violation handler (see also Section 3.6.6). However, *observe* serves another important use case: It provides the opportunity to install a logging handler to instrument an existing codebase — one that is known to run successfully in production — with contract assertions to find defects in that codebase without bringing down the entire production system upon contract violation. Other use cases for the three different semantics are discussed in [P2877R0].

[P2877R0] was approved by SG21 at the WG21 meeting in Varna in June 2023, and the *ignore*, *observe*, and *enforce* evaluation semantics were added to the Contracts proposal.

SG21, Varna, 2023-06-13, Poll 1

Modify the Contracts MVP as proposed by P2877R0 as presented.

SF	F	N	A	SA
10	11	4	3	1

Result: Consensus

The Contracts design was based on this set of three possible evaluation semantics until the WG21 meeting in Tokyo in March 2024. At that meeting, developers of `libc++` provided some feedback, which can be found in [P3191R0], on the Contracts design. In particular, they found that enforcing a contract violation via calling the contract-violation handler, constructing a `contract_violation` object to pass into the handler, and so on (see Section 3.5.10) was not a scalable approach in scenarios where code size must be kept small and termination must happen as quickly as possible to leave the smallest possible attack surface. In particular, if a contract predicate was found at run time to not evaluate to `true`, a conforming implementation should be allowed to immediately terminate the program via a `__builtin_trap` instruction, which is the approach `libc++` chose for their library hardening (see [P3471R4]).

This feedback resulted in adding a fourth evaluation semantic to the Contracts proposal. This fourth semantic does not call the contract-violation handler on violation but instead proceeds to immediately terminate the program.

SG21, Tokyo, 2024-03-21

Add a contract semantic to P2900R6 where the predicate is evaluated, on contract violation no contract-violation handler is invoked, and the program is stopped in an implementation-defined way.

SF	F	N	A	SA
23	10	0	0	0

Result: Consensus

This new, fourth evaluation semantic initially did not have a name; it was informally called the "Louis semantic" after the first author of [P3191R0]. After much discussion, which is captured in [P3226R0], and after considering dozens of possible names, SG21 finally adopted the name *quick-enforce* for this new semantic. This name is somewhat consistent with `std::exit()` vs. `std::quick_exit()`, both of which terminate the program, but the former executes more clean-up code than the latter.

SG21, Teleconference, 2024-04-18, Poll 2

Adopt the name `quick_enforce` for the new, fourth evaluation semantic added via P3191R0 to the Contracts MVP, as proposed in P3226R0.

SF	F	N	A	SA
5	11	1	1	0

Result: Consensus

An alternative set of names for the now four semantics was proposed in [P3238R0]. In particular, that paper sought to draw a connection between the new *quick-enforce* semantic and the concept of *erroneous behavior* from [P2795R5]. However, this alternative direction was not received favorably by SG21.

SG21, Teleconference, 2024-05-09

For the Contracts MVP, rename the evaluation semantics *ignore*, *observe*, *enforce*, and *quick_enforce* to *ignored*, *observed*, *enforced*, and *erroneous*, respectively, as proposed by P3238R0.

SF	F	N	A	SA
1	0	4	11	1

Result: Consensus against

Finally, another paper, [P3316R0], proposed to replace the *ignore* semantic with a slightly different semantic called *promise*. The proposed new semantic sits somewhere between *ignore* and *assume* in that it assumes not that the predicate must evaluate to `true` but that a predicate has been established as `true` *after* the assertion. The purpose of this semantic is to prevent an optimizer from seeing that a failed check would abort the program and hence optimizing on that assumption, i.e., to make *promise* swappable with checked semantics at link time without affecting the optimization of the surrounding code. This direction, however, was not received favorably by SG21.

SG21, St. Louis, 2024-06-26, Poll

We want to spend more time considering the *promise* semantic proposed in P3316R0 for P2900.

SF	F	N	A	SA
0	1	1	13	6

Result: Consensus against

SG21, St. Louis, 2024-06-26, Poll

We want to spend more time considering the *promise* semantic proposed in P3316R0 for Contracts, as a post-MVP extension.

SF	F	N	A	SA
1	5	10	4	0

Result: No consensus

As evidenced by the history of contract-evaluation semantics, the ability to check a contract assertion and terminate the program on contract violation is an essential part of Contracts in C++ and has been part of all Contracts for C++ proposals so far. However, another design aspect of the evaluation semantics that underwent many changes in the last two decades is the question of *how* the program shall be terminated in this case.

In the first Contracts for C++ proposal, [N1613], and all following proposals in Phase One, the default mode of termination was `std::terminate`. In [N3604], and for the remainder of Phase Two, the default mode of termination was changed to `std::abort`, without an explicit motivation given for this choice.

At the beginning of Phase Three, [N4110] proposed that the default termination mode should again be `std::terminate`, again with no explicit motivation given. [P0147R0] concluded that, in the absence of any knowledge of the program development environment, the default should be to terminate the program "quickly but safely" and that "the mechanism for that termination is `quick_exit`." Later, the Contracts Merged Proposal, [P0246R0], again reverted this default to `std::abort`, again without explicit motivation; this specification remained in place until the end of Phase Three when the C++2a Contracts feature was removed from the C++20 Working Draft.

In Phase Four, [P2182R1] first suggested that the Standard should not prescribe whether `std::abort` or `std::terminate` would be called at the end of the contract-violation handler. The first iteration of the proposal in that phase, [P2388R0], followed this suggestion and did not specify the exact mode of termination; instead, it proposed that "the program exits and an implementation-defined form of the status *unsuccessful termination* is returned." The paper *recommended*, non-normatively, that the mode of termination should be a call to `std::abort`. In revision [P2388R3], this recommendation was changed again to *require* that a contract violation results in a call to `std::abort` (while simultaneously, throwing an exception from the predicate would call `std::terminate`; see Section 3.5.6).

[P2466R0] argues that `std::abort` is the correct choice due to the ability to send `SIGABRT` to the environment.

[P2811R7] proposed that `std::abort` be changed again to an implementation-defined mode of termination. In particular, the paper encouraged the inclusion of additional guards protecting against user code — such as a `SIGABRT` handler being called from `std::abort` or a termination handler being called from `std::terminate` — being called after a contract violation since such code could intervene in the contract-violation handling process and then recursively violate another contract (see also 3.6.4).

In subsequent revisions, the termination mode was specified as implementation-defined, but the specification of what actually happens was closely following the description of what `std::abort` does with the exception of no actual call to `std::abort`. This modification was made for two reasons: first, to prevent a `SIGABRT` handler from intervening and, second, to allow the `<contracts>` header to be freestanding.

During SG21's review of a draft of [P2900R4], a compiler implementer observed that allowing a `SIGABRT` handler could actually be useful and that `std::abort` had been made freestanding in the meantime. The original motivation to *not* call `std::abort`, therefore, no longer applied, and SG21 voted to change the termination mode of *enforce* back to calling `std::abort`.

SG21, Teleconference, 2024-02-15, Poll 3

Change termination mode to using `std::abort`?
Result: No objections to unanimous consent

However, [P3191R0] made clear that compiler vendors need to have more flexibility for the termination mode. In particular, [P3191R0] argued why, in safety-critical scenarios, the implementation may choose to call platform-specific intrinsics, such as `__builtin_trap` or `__builtin_verbose_trap`, to perform the termination, rather than to call any Standard Library function, yet in other scenarios, calling `std::terminate` or `std::abort` may be better choices.

Therefore, the new *quick-enforce* semantic introduced by [P3191R0] was defined from the start with an implementation-defined termination mode. SG21 further decided to make the *enforce* semantic consistent with the *quick-enforce* semantic in this regard.

SG21, Tokyo, 2024-03-21, Poll

In P2900R6, when the contract-violation handler returns normally and the semantic is *enforce*, rather than calling `abort()`, the program should be stopped in an implementation-defined way.

SF	F	N	A	SA
17	13	1	1	0

Result: Consensus

At that point, a particular termination mode was not even being given as a recommendation; instead, the termination mode was completely left up to the implementation. A suggestion to add such a recommendation was rejected.

SG21, Tokyo, 2024-03-21, Poll

In P2900R6, add a recommended practice, that the enforce semantic should call `std::abort()`.

SF	F	N	A	SA
2	9	15	4	0

Result: No consensus

The topic of termination mode came up again during EWG design review at the WG21 meeting in St. Louis in June 2024. A member suggested that leaving the termination mode implementation-defined appears like a hole in the design and that if SG21 wants to allow different options for the termination mode, these allowed options should be enumerated and offered via distinct semantics. However, EWG was against such a change and deemed the choice to leave the termination mode of both *enforce* and *quick-enforce* implementation-defined sufficient for Contracts in C++.

EWG, St. Louis, 2024-06-24, Poll

P2900R10: The method for evaluation semantics' termination should not be implementation defined for *enforce*/*quick_enforce*. A solution to remove implementation-defined might be to specify the behavior of existing semantics, and could also include specifying more evaluation semantics.

SF	F	N	A	SA
2	4	7	19	4

Result: Consensus against

Following EWG design approval, CWG began its wording review of [P2900R11]. CWG opined that this completely open specification is insufficient to determine which modes of termination are actually conforming and prevents CWG from finding the right words to specify the design. CWG acknowledged the need to allow various termination modes but asked to explicitly enumerate the

specific termination modes between which a conforming implementation is allowed to choose. This request led the authors to consider all possible termination modes in C++ and their differences and tradeoffs and to conclude that only the erroneous modes of calling `std::abort` and `std::terminate` or terminating immediately with a trap-like instruction should be allowed since no other mode makes sense conceptually or has known use cases. This proposal, alongside detailed rationale, was published in Section 1 of [P3520R0] and was approved by both SG21 and EWG.

SG21, Wrocław, 2024-11-22, Poll 1

Adopt the mode of termination changes from P3520R0 Section 1 and forward to EWG.

SF	F	N	A	SA
5	8	1	0	3

Result: Consensus

EWG, Wrocław, 2024-11-22, Poll

Poll: P3520r0 contracts, EWG supports the paper's suggested fix for item 1, which adds contract-terminated and lists specific behavior, rather than having implementation defined behavior.

SF	F	N	A	SA
11	24	5	0	0

Result: Consensus

3.5.5 Selection of Semantics

In [P2900R14], the selection mechanism for the evaluation semantic (*ignore*, *observe*, *enforce*, or *quick-enforce*) is implementation-defined. The specification does not say anything more specific about that selection mechanism. Selecting the evaluation semantic may happen at compile time (via a compiler flag, a configuration file, or any other means), at link time, at load time, or even at run time (for example, through hooks inserted into the code that can be activated by a debugger or another runtime tool). Further, different contract assertions in the same translation unit or even in the same function may be evaluated with different evaluation semantics, and even subsequent evaluations of the *same* contract assertion may be evaluated with different evaluation semantics.

This specification is a major departure from previous attempts, all used in-code annotations, per-translation unit "build modes," or a combination of both to control the evaluation semantic of a given contract assertion. [P2900R14] instead moves this control entirely to the implementation vendor.

Extensive motivation for this approach is provided in [P2877R0]. At the heart of this motivation is the design goal of the Contracts facility to enable the widest possible range of use cases by maximizing implementation freedom and thus to help solve fundamental deployment and usage needs that the proposed Contracts feature must be able to meet.

Such use cases include

- Package managers and other forms of packaged software, which should not be forced to distribute a different binary for each possible evaluation semantic

- Builds that selectively enable some contract checks while disabling others, for example checking precondition assertions but not postcondition assertions
- Builds that allow conditionally enabling or disabling particular contract checks at run time (e.g., existing tools allow the user to do this by setting an environment variable with a value that is keyed to the source location of the assertion)
- REPL (read-evaluate-print loop) interpreters and other forms of interpreters where the evaluation semantic may be controlled freely at run time
- Debuggers that hook into the contract assertions

All the above use cases hinge on properties that build modes do not provide but that implementation-defined semantics as specified in [P2900R14] do. This includes, in particular, the requirement that a conforming Contracts implementation should be allowed to enable and disable contract checks without requiring a rebuild.

Note that these requirements are not universal; platforms should be free to choose not to support them as appropriate for their user base. Our intent is to keep all such implementations as well-defined, Standards conforming, and subject to the same training and reasoning about behavior that is applicable to other C++ platforms; our intent is *not* to have embedded systems, hardened compilers, and maximally optimized deployments all needing to support the same feature sets.

Implementation strategies for different selection mechanisms are discussed in [P2877R0], [P3267R1], and [P3321R0]. The latter paper also discusses compiler flags and other options that we expect vendors to provide. At the moment, implementations of Contracts in both GCC and Clang support per-translation-unit configuration of contract semantics, with the ability to freely link translation units with arbitrary different configurations. Work is in progress to add support for choosing caller-side evaluation in GCC. The Clang implementation has added attributes to locally control semantics in code to thus facilitate testing of the Contracts feature itself.

The history of arriving at this specification for selection of semantics is closely tied to the history of the semantics themselves.

In Phase One, no mechanism was provided to turn contract checks off or otherwise control the evaluation semantic.

In Phase Two, evaluation semantics were not yet provided as first-class entities that could be named or enumerated, but the effective semantics were selected by a combination of which macro is used to spell the assertion and which "assertion-level flag" (defined as a preprocessor symbol) was used to compile the code. In [N4378], the four possible macros, `contract_assert`, `contract_assert_dbg`, `contract_assert_safe`, and `contract_assert_opt`, could be combined with one of the three assertion-level flags, `contract_assertion_level_min`, `contract_assertion_level_on`, and `contract_assertion_level_max`, which would select one of the three assertion levels, `min`, `on`, and `max`, respectively. Each such level would enable contract checking on a different set of assertion macros.

At the beginning of Phase Three, [N4110] provided the first exploration of different selection mechanisms, such as a compiler flag and an attribute or keyword around a block of code. [N4248] discussed "build modes," including the downsides of standardizing build modes, e.g., a combinatorial explosion of builds and the possibility of caller-side checking.

[N4435] was first to propose that the selection of semantics should be entirely implementation-defined. [P0147R0] proposed that compilers must have options that control the extent to which they interpret and exploit Contracts and that these options should characterize contract assertions by importance and cost, and it offered a thorough exploration of meaningful concrete options along those two axes.

[N4293] discusses different options for whether or not build modes should be standardized and mentions that, at some point, EWG had consensus that build modes should not be standardized. The Contracts Merged Proposal, [P0246R0], however, was again assuming the existence of build modes. [N4415] called these "translation modes" and first explicitly mentioned the possibility of a mode that checks only preconditions or only postconditions.

[P0380R0] changed the terminology from "build modes" to "build levels" and established the selection mechanism for C++2a Contracts [P0542R5]. In this framework, the actual evaluation semantic for a given contract assertion was determined by combining the "assertion level" of that assertion (`default`, `audit`, or `axiom`), the chosen "build level" (`off`, `default`, or `audit`), and the chosen "continuation mode" (`on` or `off`). The first two parameters were effectively selecting between a checked and an unchecked semantic, and the third parameter would select the desired checked semantic (*observe* or *enforce*).

In C++2a Contracts, the mechanism for selecting the build level was specified to be implementation-defined, and the mechanism for selecting the continuation mode was entirely unspecified. [P0542R5] further specified that programs consisting of translation units in which the build level is not the same in all translation units ("mixed mode") are conditionally supported and that no programmatic way of setting, modifying, or querying the build level of a translation unit should be possible.

[P0542R5] also normatively specified defaults for these settings: If no build level is explicitly selected, the build level is `default`, and if no continuation mode is explicitly selected, the default continuation mode is `off`. [P1769R0] argued that the C++ Standard could and should not prescribe any such defaults and that the default should instead be implementation-defined; [P1769R0] was approved by EWG.

EWG, Cologne, 2019-07-15, Poll

P1769R0 as presented

SF	F	N	A	SA
16	20	8	0	2

Result: Consensus

A fourth selection parameter, the *assumption mode* (`on` or `off`), proposed in [P1710R0], [P1711R0], [P1730R0], and [P1786R0], would additionally select the desired unchecked semantic (*assume* or *ignore*) such that both can be enabled; [P1812R0] proposed an additional tweak that would have made `axiom` a type of assertion that is always assumed regardless of the mode. However, none of these proposals were approved by EWG.

EWG, Cologne, 2019-07-15, Poll

P1730R0 (Poll 1): Add global assumption mode

SF	F	N	A	SA
17	12	15	11	4

Result: No consensus

EWG, Cologne, 2019-07-15, Poll

P1730R0 (Poll 2): Add global assumption mode, remove global continuation mode

SF	F	N	A	SA
1	4	7	19	13

Result: No consensus

EWG, Cologne, 2019-07-15, Poll

D1812R0 as presented?

SF	F	N	A	SA
2	2	7	20	9

Result: No consensus

Alternative selection parameters, so-called *roles*, were proposed in [P1332R0]. [P1334R0] proposed to instead select the desired semantic, called *contract mode* at the time, syntactically, i.e., directly on the contract assertion with the tokens `assume`, `ignore`, `check_never_continue`, `check_maybe_continue`, and `check_always_continue`, respectively. These so-called *literal semantics* were originally proposed as a mechanism *in addition* to assertion levels, build levels, and the continuation mode; however, in [P1429R3] and [P1607R1], literal semantics were instead proposed as a *replacement* mechanism for all of these. This replacement of all other semantic-selection mechanisms with literal semantics received consensus in EWG.

EWG, Cologne, 2019-07-15, Poll

P1607R0 (Poll 1): Take away build levels and continuation

SF	F	N	A	SA
10	18	8	2	7

Result: Consensus

EWG, Cologne, 2019-07-15, Poll

P1607R0 (Poll 2): Literal semantics

SF	F	N	A	SA
15	13	5	3	10

Result: Consensus

However, such a drastic design change at a very late stage in the process ultimately eroded trust in the overall design of C++2a Contracts and contributed to the decision to remove them from the C++20 Working Paper via [P1823R0], marking the end of Phase Three.

In Phase Four, papers such as [P2076R0] and [P2182R1] sought to understand the previous disagreements and pave a way forward. With regard to semantic selection mechanisms, these papers raised the questions of whether global switches that affect all contract assertions should be available and whether a mechanism for explicit per-contract semantic selection on the contract assertion itself should be present.

The first iteration of the proposal in Phase Four, [P2388R0], reintroduced the notion of build modes and proposed two such modes: *Ignore* and *Check_and_abort*, which translate to today's *ignore* and *enforce* semantics, respectively. These build modes were subsequently renamed to *No_eval* and *Eval_and_abort* in revision [P2388R3]. These build modes were meant to be valid per translation unit, with mixed mode (linking together translation units compiled with different build modes) being allowed.

After much discussion in SG21, the notion of build modes was removed by approving [P2877R0]. Build modes were replaced with the present notion of *evaluation semantics* (which had already been proposed as first-class entities at the end of Phase Three) and with the specification that the selection mechanism for the evaluation semantic is not only implementation-defined, but can also vary per contract assertion and even per evaluation of the same contract assertion to enable the fullest breadth of use cases and give compilers and tools the necessary freedom to support them.

Earlier iterations of the Contracts proposal from before the time evaluation semantics were added (i.e., before [P2388R4]) recommended that *Eval_and_abort* should be the default mode. The analogous recommendation that *enforce* should be the default evaluation semantic if nothing else has been specified by the user[21] was approved via [P2877R0] and remains in [P2900R14] today. This recommendation is non-normative and is thus an in-between position between the C++2a Contract approach of normatively mandating a default and the [P1769R0] approach previously approved by EWG that the C++ Standard should refrain from specifying what the default should be.

Concerns echoing [P1769R0] that even non-normatively recommending a default in the Standard is not meaningful were discussed in SG21, but removing the recommendation of a default from the proposal did not get sufficient consensus. (See Section 3.5.4 for the Poll 1 to which this poll's results refer.)

[21] This specification includes the understanding that optimization flags and other build configuration options qualify as "something specified by the user" even if they are not explicitly naming a desired contract-evaluation semantic.

SG21, Varna, 2023-06-13, Poll 2

Modify the Contracts MVP as proposed by P2877R0 with Proposal 5 (Default Semantic) removed.

SF	F	N	A	SA
6	10	7	2	3

Result: Weaker consensus than Poll 1

[P3582R0] raised a concern that if the selection of evaluation semantics is entirely free, adding *observed* contract assertions to a program could introduce undefined behavior where there was none previously if the defined behavior of checking one assertion relies on the success of the previous one:

```
Tool* selectTool(Tool* ta, Tool* tb)
  pre (ta != nullptr)
  pre (ta->is_configured())
  pre (tb != nullptr)
  pre (tb->is_configured())
  post (r: r != nullptr)
  post (r: r->is_configured())
{
  // not dereferencing ta or tb:
  if (preferFirst) return ta;
  else             return tb;
}
```

The remedy proposed by [P3582R0] was to specify that if one *observed* contract assertion fails, all subsequent contract assertions in that contract assertion sequence should be skipped. SG21 discussed this proposal in great detail and decided *not* to pursue it.

SG21, Teleconference, 2025-02-06, Poll

Forward P3582R0 "Observed a contract violation? Skip subsequent assertions!" to EWG for C++26.

SF	F	N	A	SA
0	0	1	13	7

Result: Consensus against

This decision was based on multiple reasons, including doubts over the validity of the example and that such a change would preclude a future extension that would allow adding an "always enforce" label to a contract assertion. (Such an extension is being proposed in [P3400R0]; the importance of such an extension for safety and security is explained in [P3500R1].) Note that if we want to use observed contract assertions as shown in [P3582R0], introducing new undefined behavior can be avoided by combining the related predicates into a single contract assertion with a logical AND operator that short-circuits if the first predicate fails:

```
Tool* selectTool(Tool* ta, Tool* tb)
  pre (ta != nullptr && ta->is_configured())
```

```
  pre (tb != nullptr && tb->is_configured())
  post (r: r != nullptr && r: r->is_configured())
{
  // ...
}
```

Note further that to prevent undefined behavior introduced by an *observed* contract assertion, a conforming implementation is free to just not offer the *observe* semantic at all on a given platform, as explicitly highlighted in both the frontmatter and the wording of [P2900R14]. In fact, the concern that some implementations (say, on some safety-critical embedded platform) might not want to offer the *observe* semantic at all was already discussed and explicitly taken into account by SG21 when the *observe* semantic was added to the Contracts paper via [P2877R0].

3.5.6 Checking the Contract Predicate

In [P2900R14], the bulk of the rules for evaluating a *checked* contract predicate originate from [P2751R1]. Detailed rationale for the design decisions behind those rules can be found in that paper; here, we give a brief summary of that rationale as well as the history of their adoption.

[P2900R14] lists the five possible situations that can arise when the value of the contract predicate is determined as part of a contract check. The first two of these are straightforward: The predicate evaluating to `true` indicates that *no* contract violation has occurred and execution should continue normally, and the predicate evaluating to `false` means that a contract violation *has* occurred and does not require special motivation. No other behavior would make sense for a contract-checking facility.

The third case deserves more consideration. The evaluation of the predicate producing no value does not necessarily mean that a violation of the plain-language contract has, in fact, occurred, but only that we failed to verify that a violation did *not* occur. The correct behavior is less self-evident in this case.

First, let us consider the situation in which predicate evaluation exits via an exception. This scenario can happen if predicate evaluation reaches some resource limit that the actual function invocation will not reach. Such situations should still be treated as a runtime contract violation, not because a violation of the plain-language contract of the actual function has occurred, but because the contract-violation mechanism allows us to defer the error to the contract-violation handler to give the user the opportunity to determine what the proper next course of action will be. The library API (see Section 3.7) provides access to the thrown exception. For example, if the exception thrown is `std::bad_alloc`, the user might decide to rethrow this exception and handle it further up the stack or to terminate the app; if the exception thrown indicates a more specific error, the user might instead decide to treat that error as a program defect in the same way as a contract check that yields `false` for the value of the predicate. For this reason, we also do not introduce a separate contract-violation handler for this situation.

The case of predicate evaluation exiting via an exception was not considered explicitly until Phase Three. The C++2a Contracts approach (see [P0542R5]) was to invoke `std::terminate` in this case. This approach is consistent with other cases in the C++ language in which an exception is being thrown in situations where it cannot be handled as usual; however, this approach precludes the

important possibility of recovering from such a situation, which our specification provides. The other possibility would be to propagate such an exception up the stack, but that choice precludes the equally important possibility, which our specification also provides, of treating such a failure to evaluate a contract predicate differently than an exception thrown anywhere else in the code.

As part of reviewing [P2570R2], [P2756R0], and [P2751R1], SG21 first explicitly considered all three options at the WG21 meeting in February 2023 in Issaquah. SG21 reached consensus that exceptions emanating from the predicate evaluation should not unconditionally call `std::terminate`, thus moving away from the C++2a Contracts design, but no consensus was reached regarding a concrete alternative.

SG21, Issaquah, 2023-02-07, Poll

In case the evaluation of a contract-checking predicate throws an exception, we want `std::terminate` to be called.

SF	F	N	A	SA
0	2	5	14	8

Result: Consensus against

SG21, Issaquah, 2023-02-07, Poll

In case the evaluation of a contract-checking predicate throws an exception, we want this to be treated as a contract violation.

SF	F	N	A	SA
7	7	3	3	7

Result: No consensus

SG21, Issaquah, 2023-02-07, Poll

Poll: In case the evaluation of a contract-checking predicate throws an exception, we want to propagate that exception out of the contract check.

SF	F	N	A	SA
5	5	4	3	10

Result: No consensus

Due to this lack of consensus on an alternative, [P2852R0] proposed, as a stopgap until a decision can be made, that the behavior be unspecified.

The concrete solution now in [P2900R14] was adopted by SG21 a few months later, after discussing [P2811R7], [P2834R0], and [P2858R0] and after considering the wider design space for the interaction between Contracts and exceptions, which includes the desired behavior for exceptions thrown from the predicate evaluation and exceptions thrown from the contract-violation handler (a separate case discussed in Section 3.5.9) as well as for the interaction between Contracts and `noexcept`.

SG21, Teleconference, 2023-05-18, Poll 2

Throwing an exception from the evaluation of a contract-checking predicate shall be treated as a contract violation, and invoke the violation handler, regardless whether the function is declared `noexcept`, as proposed in P2811.

SF	F	N	A	SA
8	7	2	0	1

Result: Consensus

The suggestion to use a separate contract-violation handler for this case came up during LEWG's review of the Contracts proposal at the WG21 meeting in March 2024 in Tokyo (see also [P3198R0]); SG21 considered and rejected that suggestion.

SG21, Tokyo, 2024-03-21, Poll

Use two different handlers for a contract predicate that evaluates to `false` and a contract predicate whose evaluation exits via an exception, respectively.

SF	F	N	A	SA
0	3	3	8	11

Result: Consensus against

At the same LEWG review, a suggestion was made to change the terminology "contract violation" and "contract-violation handler" to better reflect the nature of this event. This suggestion, too, was rejected by SG21; the existing terminology was deemed sufficiently clear and established.

SG21, Tokyo, 2024-03-21, Poll

We want to rename the terms "contract violation" and "contract violation handler" and their associated library names in P2900 to some other terms that reflect that the situations described by them do not necessarily represent a violation of the plain-language contract of a function.

SF	F	N	A	SA
0	7	9	9	3

Result: No consensus

In the lead-up to the February 2025 WG21 meeting in Hagenberg, concerns over the decision to treat a contract check that exits via an exception as a contract violation were raised again in [P3506R0] and [P3573R0]; those papers claimed that this design would impose overhead and hinder composition and recommended to instead unconditionally unwind the stack in that event. A rebuttal of these claims was provided in [P3591R0], which argues that the design of [P2900R14] in this area is correct as is. EWG discussed this matter at the Hagenberg meeting; as supplementary material for that discussion, [P3626R0] provided the wording diff that would need to be applied to [P2900R14] to change the design to follow the recommendation in [P3506R0] and [P3573R0]. EWG did not have consensus for such a change, and as a result, [P2900R14] proceeded as is.

EWG, Hagenberg, 2025-02-11, Poll

P2900: unconditionally unwind exceptions when they leave predicate evaluation.

SF	F	N	A	SA
12	18	11	15	7

Result: No consensus

For the fourth case, where control never returns to the purview of the contract-checking process (predicate evaluation calls `longjmp`, terminates the program, enters an infinite loop or suspends the thread indefinitely, and so on), we have fewer options because treating these cases as a contract violation is not technically feasible. Calling `longjmp` was first considered in [P2388R1] and specified to be undefined behavior; however, this design goes against [P2900R14]'s Design Principle 13, Explicitly Define All New Behavior. [P2751R1] and [P2852R0] proposed to define that all such situations, including `longjmp`, follow normal C++ rules for expression evaluation, i.e., "You get what you get." This proposal was adopted by SG21.

SG21, Issaquah, 2023-02-08, Poll 1

When the evaluation of a contract-checking predicate exits the control flow other than by returning or throwing (for example, `abort` or `longjmp`), the behavior is as if it were not in a contract-checking predicate.

SF	F	N	A	SA
19	11	0	1	0

Result: Consensus

For the case in which the predicate is evaluated during constant evaluation and the predicate expression is not a core constant expression, see Section 3.5.12. For the case in which the predicate evaluation has undefined behavior, see Section 3.6.1.

3.5.7 Elision, Duplication, and Evaluating in Sequence

The allowance in [P2900R14] that the implementation may *elide* and *duplicate* contract-assertion evaluations is a conscious departure from the model of `cassert` and similar macro-based facilities to thus enable additional usage and deployment scenarios not currently served by those facilities.

Arguably the most important of these scenarios is the ability to independently configure translation units to check precondition and postcondition assertions either caller-side or callee-side and then to link those translation units together, which — at least if the implementation wishes to avoid ABI changes (see also Section 3.5.11) might result in checks happening twice, once caller-side and once callee-side. Other scenarios include a conforming testing mode to evaluate contract assertions an arbitrary number of times (including more than twice) to identify destructive side effects (see Section 3.5.8) and the desire to elide unnecessary evaluations to reduce the runtime impact of contract checks.

Detailed motivation for elision and duplication can be found in several papers. The first such paper is [P2751R1], which was adopted by SG21 and introduced elision and duplication to the Contracts

proposal. An in-depth analysis of the available design space and the tradeoffs involved is available in [P3228R1]; a similar, independent analysis is available in [P3270R0]. Note that this topic is also closely related to that of side effects in contract predicates (see Section 3.5.8).

Elision and duplication of contract assertions cannot be made to work as desired if each contract-assertion evaluation is considered in isolation; instead, we identify ordered sets of contract assertions that are evaluated together (*evaluated in sequence*) and provide the freedom to repeat contract assertions when evaluating those sets of contract assertions. One such reason is that we need to specify what happens when duplication — e.g., due to caller- and callee-side checking happening simultaneously — happens on a function call that has multiple precondition or postcondition assertions associated with it, and the specified semantics need to be implementable and map to real-world deployment scenarios. We also need to make sure that eliding contract assertions does not lead to unexpected situations, such as precondition assertions being checked in the wrong order (since one can rely on the other). The basic framework for all these evaluation rules is explained in [P2751R1]; a later refinement was introduced in Item 4 of [P3520R0] to scope repetition to only the evaluation of preconditions, postconditions, and consecutive assertion statements.

Elision of contract assertions was first considered in Phase One; [N1669] specified that if the compiler can determine that the condition of an assertion is `true`, it may choose not to evaluate that assertion. Duplication was first considered in the early stages of Phase Three; [P0247R0] notes that evaluating some checks twice seems tolerable and, in general, unavoidable. C++2a Contracts did not allow elision and duplication explicitly but did so implicitly since both are observable (at least within the C++ Abstract Machine) only if the predicate evaluation has side effects and if such side effects were specified to be undefined behavior, meaning that anything could happen.

This topic was considered in depth at the WG21 meeting in February 2023 in Issaquah. SG21 considered [P2750R2], which explored the different possible semantics; [P2751R1], which was proposing the current model; and [P2756R0], which instead proposed the `cassert` model where every *checked* assertion is evaluated exactly once. The model proposed in [P2751R1] was adopted.

SG21, Issaquah, 2023-02-07, Poll

Do we want to require that a contract-checking predicate is evaluated exactly once in `eval_and_abort` mode?

SF	F	N	A	SA
2	2	0	9	15

Result: Consensus against

SG21, Issaquah, 2023-02-07, Poll

Do we want to allow that a contract-checking predicate is evaluated zero or more times in `eval_and_abort` mode?

SF	F	N	A	SA
14	10	0	1	2

Result: Consensus

SG21, Issaquah, 2023-02-08, Poll

We want to adopt the rules on reordering the evaluation of contract-checking predicates as described in P2751R0 proposal 3.4, contingent on a clarification that this reordering can be equivalently formulated in terms of elision of evaluations inside repetitions of the given sequence of predicates.

SF	F	N	A	SA
12	5	3	1	0

Result: Consensus

When the Contracts proposal was design-reviewed in EWG, the concern was raised that departing from the `cassert` model where every *checked* assertion is evaluated exactly once may be surprising to users and makes migration from macro-based assertion facilities to Contracts more difficult. This concern was well known in SG21 when the Contracts facility was designed, but the tradeoffs involved were evidently weighed differently by different people. An EWG poll on this matter resulted in a split room that threatened the consensus on the Contracts design.

EWG, Tokyo, 2024-03-20, Poll 7

P2900r6: contracts should not be able to evaluate preconditions/postconditions/assertions more than once per invocation.

SF	F	N	A	SA
13	8	15	10	8

Following this poll, the discussion around elision and duplication was reignited. Two refinements to the duplication rules were introduced in [P3119R1]. Having no upper bound on the number of duplications was argued to be a problem; a contract assertion performing cumulative addition of signed integers could be said to always have undefined behavior because if an unlimited number of duplications is allowed, such an addition will eventually overflow. The first refinement added such an upper bound, with the actual number being implementation-defined to avoid imposing a concrete number, such as two that would constrain implementation freedom and preclude some use cases such as the "repeat all assertions N times" testing mode. The second refinement reconfirmed in wording that *no* repetitions is the recommended default and the expected most common scenario. Both refinements were adopted.

SG21, Teleconference, 2024-05-09, Poll

For the Contracts MVP, introduce an implementation-defined upper bound on the number of times a given contract assertion may be repeated in a contract assertion sequence, and add a recommended practice that an implementation should provide an option to perform a specified number of such repetitions and that, by default, no repetitions should be performed, as proposed by P3119R1 Proposal 3.B.

SF	F	N	A	SA
5	8	0	3	1

Result: Consensus

However, despite these refinements, concerns over elision and duplication remained. Counterproposals were published to revert to the `cassert` model of exactly one evaluation, either for `contract_assert` only ([P3257R0]) or for all three kinds of contract assertions ([P3281R0]). Both counterproposals were considered by SG21 but failed to get consensus.

SG21, Teleconference, 2024-05-16, Poll 3

For the Contracts MVP, specify that the predicate of a checked assertion statement (`contract_assert`) is evaluated exactly once, as proposed in P3257R0 Proposal 2.

SF	F	N	A	SA
8	4	0	4	3

Result: No consensus

SG21, Teleconference, 2024-05-16, Poll 4

For the Contracts MVP, specify that the predicate of a checked contract assertion of any kind (`pre`, `post`, and `contract_assert`) is evaluated exactly once, as proposed in P3281R0.

SF	F	N	A	SA
4	2	1	4	10

Result: Consensus against

The question was settled in EWG at the next WG21 meeting in June 2024 in St. Louis, which reconfirmed the allowance to repeat the contract checks, potentially even more than twice.

EWG, St. Louis, 2024-06-24, Poll

P2900r7: contracts should not be able to evaluate preconditions/postconditions/assertions more than once per invocation.

SF	F	N	A	SA
3	2	2	17	11

Result: Consensus against

EWG, St. Louis, 2024-06-24, Poll

P2900r7: contracts should not be able to evaluate preconditions/postconditions/assertions more than twice per invocation.

SF	F	N	A	SA
1	8	4	17	5

Result: Consensus against

The refinement that such repetition should be scoped to only the evaluation of the preconditions of a given function or of its postconditions or of consecutive assertion statements was introduced in Item 4 of [P3520R0] as a response to concerns raised during wording review of [P2900R11]. (See [P3520R0] for a detailed explanation of the issue and the resolution.) This resolution was approved by both SG21 and EWG.

SG21, Wrocław, 2024-11-22, Poll 4

Adopt the changes to the definition of a contract assertion sequence from P3520R0 Section 4.

SF	F	N	A	SA
8	6	1	0	0

Result: Consensus

EWG, Wrocław, 2024-11-22, Poll

P3520r0 contracts, EWG supports the paper's suggested fix for item 4, which limits contract assertion sequences.

SF	F	N	A	SA
14	10	8	0	0

Result: Consensus

3.5.8 Predicate Side Effects

As discussed in the previous section, [P2900R14] allows elision and duplication of contract predicate evaluations. The unavoidable consequence of this design is that, if those predicates have side effects when evaluated, these side effects can also be observed multiple times or not at all. This consequence is the main reason why elision and duplication were contentious when introduced: They affect the observable behavior of the program. In [P2900R14], side effects from contract predicates cannot be relied upon, similarly to side effects from copy constructors, which cannot be relied upon due to copy elision. This behavior might be surprising to users who might instead expect the `cassert` semantics where we can reason about any side effects happening exactly once when the assertion is checked.

In a Contracts facility that allows elision and duplication, we could deal with observable side effects in predicate evaluations in three hypothetical ways.

1. The side effects occur an unspecified number of times, which might be zero ("You get what you get").

2. Evaluating a predicate with side effects is undefined behavior.

3. Predicates with side effects are ill-formed.

[P2900R14] has chosen the first option because the other two are nonviable.

Making side effects undefined behavior goes against [P2900R14]'s Design Principle 13, Explicitly Define All New Behavior, and is actively user hostile: Adding an innocuous print statement, e.g., for debugging purposes, to a function anywhere in our code could lead to undefined behavior if that function happens to be called from a contract predicate. This motivation is explored in more detail in [P1670R0], the paper that first presented the side effects model that is now part of [P2900R14].

Making predicates with side effects (either outside their cone of evaluation or any side effects at all) ill-formed was suggested several times during the development of the Contracts feature but is equally a nonviable route. To understand this decision, we must distinguish the concept of *core language side effects* (I/O functions, modifications of volatile glvalues) from the concept of *destructive side effects* as defined in [P2900R14], i.e., side effects that affect whether the program can meet its plain-language contract. This distinction is discussed in much more detail in [P2712R0], where it was originally introduced, and in [P2751R1] and [P3228R1]. In short, having a core-language side effect is neither necessary nor sufficient for a predicate to qualify as destructive since the destructiveness of a predicate depends on user intent, which is unknowable to the compiler. For example, a print statement, while a core-language side effect, would be a nondestructive side effect in any program where what is printed to `stdout` is not part of that program's plain-language contract and could be useful in many of these programs; conversely, evaluating an expression with no core-language side effects whatsoever can still lead to a destructive side effect if that evaluation breaks the complexity guarantees of the function.

Additionally, making ill-formed predicates that have core-language side effects outside of their cone of evaluation requires drastically reducing the set of allowed expressions. For example, we could not call a function in a contract predicate unless the compiler could somehow prove or assume that every such function is itself also free of side effects. Even a contract assertion as simple as `pre (!vector.empty())` would, therefore, not compile, rendering such a Contracts facility not practically useful. Note that we cannot in general prove whether a C++ expression is free of side effects; doing so is akin to solving the halting problem. Therefore, the only viable specification and implementation strategy would be to constrain contract predicates to the set of expressions for which such a proof is possible in all cases.

In addition, we have no specification and implementation experience with such an approach. `constexpr` functions are frequently cited as an example where provably side-effect-free evaluation has been successfully specified and implemented, but constant evaluation free from side effects is a fundamentally different problem; during constant evaluation, all input parameters are known when the constant expression is actually evaluated, and to reject predicates that could potentially have side effects would require constructing such a proof for all possible inputs.

Therefore, [P2900R14] allows side effects in contract predicates but does not guarantee whether or how many times these side effects will be observable, meaning that we cannot rely on them and that predicates with side effects are generally discouraged. This approach is easy enough to teach and consistent with what we have always been teaching with regard to using assertions; i.e., they should be checking the state of the program, not modifying it.

Though side effects can occur multiple times or not at all, one important guarantee is that side effect elision and duplication in [P2900R14] is always *atomic*: Either all or none of the side effects of a particular predicate evaluation can be elided or duplicated, and eliding or duplicating some side effects of that predicate but not others is not permitted. This guarantee is important because it allows us to write predicates that rely on RAII (e.g., lock a mutex, check a variable, and then unlock it) and reason about the mutex always being unlocked if it was locked while evaluating the predicate.

The question of how to deal with side effects in contract predicates came up regularly during the history of standardizing Contracts for C++. While all proposals agree that side effects are generally undesirable, many earlier proposals avoided spelling out what that actually means in practice (whether they should be UB, ill-formed, or something else) and instead used vague wording. According to [N4110], contract assertions "should not have" side effects; in [N4293], "no side effects should be allowed"; in [N4415], contract assertions " should be side effect free"; and in [P0246R0], contract assertions "are assumed to have no side effects." Other papers were more concrete: [N4378] proposed that side effects should be ill-formed, no diagnostic required (IFNDR), and [N4435] proposed that evaluating a predicate that is not pure (i.e., free of side effects) should be undefined behavior.

To our knowledge, the first time the matter was discussed in depth in WG21 was at the March 2018 WG21 meeting in Issaquah. The following polls were taken.

EWG, Jacksonville, 2018-03-15, Poll

Do what is proposed here, making ill-formed no diagnostic required?

SF	F	N	A	SA
5	8	8	3	5

EWG, Jacksonville, 2018-03-15, Poll

Same as above with undefined behavior in all cases?

SF	F	N	A	SA
6	7	8	5	3

EWG, Jacksonville, 2018-03-15, Poll

Same as above making them well-formed?

SF	F	N	A	SA
1	5	7	9	6

Since those polls did not yield consensus for any of the options, an AB poll was taken to decide between the first solution (IFNDR) and the second solution (UB).

EWG, Jacksonville, 2018-03-15, Poll

First or second solution?

First	Second
8	16

Thus, side effects in contract predicates were made undefined behavior. For C++2a Contracts, this was implemented in wording in revision [P0542R2] but not being mentioned in the paper's text; revision [P0542R4] clarified that side effects were indeed undefined behavior but only outside the cone of evaluation of the contract predicate. Later, this choice was recognized as surprising and user hostile. A better model, to make side effects well defined but elidible, was first proposed in [P1670R0].

Early versions of the Contracts proposal adopted that model but were inconsistent on the details. [P2388R0] proposed that side effects are discouraged but allowed and that the implementation is allowed to discard or duplicate all side effects of the evaluated predicate as long as doing so does not affect the value returned by the expression. Revision [P2388R3] relaxed this rule and also allowed discarding side effects per *subexpression* of a predicate.

In [P2461R1], this choice was recognized as a design flaw because it complicated reasoning about and relying on the semantics of a predicate. For example, if the predicate uses a RAII-style object, then evaluating the constructor but not the destructor makes no sense. The paper proposed that either all or none of the predicate side effects should be elided or duplicated.

Finally, [P2751R1] proposed the complete model that is part of [P2900R14] today. Though SG21 adopted that model, this particular design decision was never explicitly polled in SG21, leading some to believe that side effects were still UB. A poll was taken to confirm consensus for the model proposed in [P2751R1].

SG21, Teleconference, 2023-12-07, Poll 2

For the Contracts MVP, a contract predicate is a C++ expression that contextually converts to `bool` and, when evaluated (including the conversion to `bool`), follows the normal C++ rules for expression evaluation; the Contracts MVP does not introduce any new undefined behaviour into the evaluation of such predicates.

SF	F	N	A	SA
10	7	1	0	0

Result: Consensus

An alternative direction was proposed in [P2680R0]: that contract predicates that have side effects outside their cone of evaluation should be made ill-formed by default, a concept subsequently called *strict predicates*. Even more controversially, the paper suggested that strict predicates could and should also be made free of undefined behavior when evaluated. The paper did not, however, provide a proper specification for strict predicates, nor did it provide a plausible implementation strategy or an explanation of how such a restricted set of contract predicates could be made usable in practice. This paper triggered a series of follow-up papers, discussions, and polls in several different WG21

groups; all of which are summarized in Section 3.6.1. Ultimately, the idea of strict predicates was not pursued further.

3.5.9 The Contract-Violation Handler

Providing a user-replaceable contract-violation handler is an important piece of functionality proposed in [P2900R14] because it allows the user to implement their own handling strategy for contract violations. The default contract-violation handler is a useful fallback, but no possible handling strategy is correct for all users on all platforms, and having a consistent and portable way to customize behavior will greatly increase the utility (and thus, in some views, the viability) of a Contracts facility for a wider range of users.

Making that contract-violation handler *global*, such that a contract violation in any component of the program will end up calling the same handler, is useful because, in general, the decision on how to report errors to entities outside the scope of the abstract machine is one that should be done per application, not by each individual library or translation unit. In addition, having a global violation handler defined by the Standard allows violations detected by many distinct third-party libraries to funnel diagnostics to a single destination defined by the application owner, which is not a piece of functionality that would be available when each library uses its own bespoke macro-based assertion facility. If the user wishes to have dedicated contract-violation handlers for particular scenarios, they can always implement and call such handlers from the global one.

The suggestion has been made, a few times, that we could introduce different contract-violation handlers for different failure modes; e.g., have one for the case in which the predicate evaluates to `false` and another one for the case in which predicate evaluation exits via an exception. However, this design is not scalable. The possible failure modes are an open set: In future extensions, we are likely to add more failure modes that can be detected via contract checks (see [P3100R1]), and we want to avoid adding what would become an essentially unbounded number of different handlers to the C++ language.

Installing a user-defined contract-violation handler at *compile time* is not feasible because different translation units of the program might be compiled at different times and with different toolchains; installing a handler at run time poses an unacceptable security risk. Therefore, we chose to install the contract-violation handler at *link time* by making it a replaceable function in the same way as global `operator new` and `delete` already are. The only difference is that the Standard Library provides only a *definition*, not a *declaration*, of the default contract-violation handler; this lack of declaration is required to install a user-defined contract-violation handler with a throwing specification different from that of the default handler (see also Section 3.6.6).

On some particularly specialized platforms, the ability to replace the contract-violation handler might be considered an unacceptable security risk even if it happens at link time. Therefore, the ability to supply a user-defined contract-violation handler is only conditionally supported.

The signature of the contract-violation handler is designed to accommodate the associated proposed Standard Library API (see Section 3.7).

The design of the contract-violation-handler, the motivation and tradeoffs of this design, and the expected implementation and usage patterns are described in much more detail in [P2811R7]. Additional motivation for having a user-defined contract-violation handler is provided in [P2838R0].

The history of contract-violation handling is as old as the history of the Contracts facility itself. One particularly noteworthy innovation of [P2900R14] compared to earlier Contracts proposals is that it decouples the contract-violation handler from the strategy for termination (or continuation) on contract violation, which is instead determined by the evaluation semantic (see Section 3.5.4). Earlier Contracts proposals usually folded this strategy into the contract-violation handler, where the termination would be performed.

The first proposal for C++ Contracts in Phase One, [N1613], allowed defining a user-defined contract-violation handler directly in code on each contract assertion; if omitted, that assertion was said to be "defaulted"; i.e., it would call a default contract-violation handler, which would call `std::terminate`. In the next revision [N1669], defining a global contract-violation handler per kind of contract assertion was possible: `set_precondition_broken_handler`, `set_postcondition_broken_handler`, and so on.

The first proposal for C++ Contracts in Phase Two, [N3604], allowed the possibility of defining a contract-violation handler at run time via `set_violation_handler`. Revision [N3997] added the possibility of setting a thread-local contract-violation handler via `handle_contract_violation_guard`.

At the beginning of Phase Three, several papers explored the design space of user-defined contract-violation handlers again. [N4110] concluded that this feature should be provided. [P0147R0] recognized the infeasibility of providing user-defined contract-violation handlers at compile time and proposed to instead do so at run time, like some of the earlier proposals. [P0246R0] first proposed link-time installation as a replaceable function like `operator new`.

At the WG21 meeting in November 2018 in San Diego, EWG did not have consensus to require a user-defined contract-violation handler.

EWG, San Diego, 2018-11-07, Poll

Require implementations to provide a way for users to designate a violation handler

SF	F	N	A	SA
1	3	8	4	9

Nevertheless, C++2a Contracts provided this feature from [P0380R0] forward. Instead of the link-time replaceable function approach, the mechanism to install a user-defined contract-violation handler was merely specified to be "implementation-defined."

In Phase Four, initially no contract-violation handler was provided due to a desire to keep the proposal minimal, although the design space was explored further in some depth in [P2339R0].

[P2838R0] recognized that *not* providing this functionality was a serious impediment to the adoption of Contracts. The paper also noted that a way to control the behavior without recompilation was a must; otherwise, the approach does not scale. After discussing the matter, SG21 had consensus to add a a user-defined contract-violation handler and adopt the link-time replaceable function approach.

SG21, Teleconference, 2023-04-20, Poll

The Contracts MVP should have a standard interface for a link-time replaceable violation handler.

SF	F	N	A	SA
10	5	1	0	1

Result: Consensus

The detailed specification for this feature was proposed in [P2811R7]. This proposal was adopted by SG21 with only a minor change that concerns the Standard Library API (see Section 3.7).

SG21, Varna, 2023-06-15, Poll 2

Amend the Contracts MVP as proposed by P2811R5 with the header name changed as per Poll 1.

SF	F	N	A	SA
13	5	4	0	0

Result: Consensus

[P3191R0] observed that weak linkage, which is the known implementation strategy for replaceable functions, can trigger a nonbenign ODR violation but recognized that this concern not actionable since we currently have no better way of providing replaceable functions.

3.5.10 The Contract-Violation Handling Process

The contract-violation handling process described in the corresponding section in [P2900R14] follows directly from the definition of the four contract-evaluation semantics (see Section 3.5.4), the rules for how the predicate is evaluated (see Section 3.5.6), the definition of the contract-violation handler itself (see Section 3.5.9), and the shape of the proposed Standard Library API (see Section 3.7).

Further details and motivation about the details of this process are given in the papers — primarily [P2811R7], which is also the origin of the pseudocode listing in [P2900R14] — referenced in those sections.

3.5.11 Mixed Mode

The corresponding section in [P2900R14] contains a detailed discussion of mixed mode as well as the allowed behaviors and plausible implementation strategies. Further discussion of mixed mode can be found in [P3321R0] and [P3267R1], which deal with the interaction between Contracts and their possible implementation strategies with tooling.

The need for mixing translation units with different build modes was first recognized in Phase Three; however, [P0542R5] explicitly did not *require* implementations to support such mixing and instead made mixed mode conditionally supported with implementation-defined semantics.

With the adoption of [P2877R0], all concepts of a build mode were explicitly removed from the specification of Contracts, and instead arbitrary mixing of configurations was allowed for

implementations; though support for such mixing is not actually mandated, all aspects of contract-assertion configuration are now in the hands of the vendors.

Once this design was established, the question of what happens when contract assertions on inline functions are compiled with different evaluation semantics in different translation units and then linked together generated a considerable amount of discussion. While [P2900R14] lists zero overhead on *ignored* assertions, compatibility with current ABIs and C++ toolchains, and full flexibility in the choice of evaluation semantics as principles of its design, satisfying all three simultaneously turns out to be impossible in this specific case. A conforming implementation of [P2900R14] inevitably needs to give up on one of the following items as a tradeoff:

- *Deterministic* behavior in mixed mode; you are guaranteed to get one of the evaluation semantics chosen in one of the TUs, but you do not know which one
- Zero overhead on *ignored* assertions, by choosing the evaluation semantic *at run time* in some deterministic fashion
- Compatibility with current linkers, by exposing the chosen evaluation semantic in each TU through an ABI extension and adding logic to the linker to make a deterministic choice (possibly based on user configuration or based on a rule such as "always pick the safer semantic")

While the impossibility of satisfying all three requirements simultaneously was cited as a problem by several WG21 members and was listed as a concern in [P3573R0], note that it is an inevitable consequence of any possible design that adds assertions to the language that can be turned on and off, *not* a flaw in [P2900R14]'s design, and thus it cannot be remedied by making different design choices. The underlying problem of how to deal with a function that is compiled down to different code in different translation units is unrelated to Contracts, has existed before Contracts, and is not made worse by Contracts.

Note further that because a difference in evaluation semantic does not constitute a violation of the ODR, [P2900R14]'s nondeterministic behavior in mixed mode ("you get some semantic") is still a significant improvement over IFNDR (which is what you get with today's macro-based Contracts facilities).

Nevertheless, during discussion of [P3573R0] and [P3591R0] in EWG at the February 2025 WG21 meeting in Hagenberg, the opponents of [P2900R14] argued that its mixed-mode semantics are effectively still an ODR violation in spirit (even if not by the letter of the Standard, because different contract evaluation semantics are *not* differences in the token stream of a function's definition) and that something should be done about it (without actually proposing even a hypothetical solution).

EWG took a deliberately vaguely worded poll to gauge the interest of the room to "do something about it"; the result was consensus against, so the design of [P2900R14] remained as it is.

EWG, Hagenberg, 2025-02-11, Poll

P2900: add ODR to contracts.

SF	F	N	A	SA
6	5	10	25	17

Result: Consensus against

3.5.12 Constant Evaluation

That contract assertions should also be evaluated during constant evaluation is a reasonable expectation. In the following program,

```
constexpr int f(int n)
  pre (n > 0) { /* ... */ };

std::array<int, f(-1)> arr = {};
```

we would like to have a compiler error that points directly at the violated precondition assertion of `f`. Though this basic motivation is clear, the actual specification for constant evaluation of contract assertions needs to consider many technical details, such as trial evaluation. The current specification in [P2900R14] is described and motivated in the corresponding section there; it was adopted from [P2894R2], where an even more comprehensive description and motivation can be found.

One significant innovation of [P2900R14] is that the *same* contract assertion can be evaluated either during constant evaluation or at run time, depending on when control flow reaches it, which is not something that `static_assert` can provide.

Another innovation is that, analogous to runtime evaluation, compile-time evaluation can happen with different evaluation semantics depending on what the user wants to do; *enforce* and *quick-enforce* (which do the same during constant evaluation because no such thing as a compile-time contract-violation handler exists) make the program ill-formed, *observe* merely issues a diagnostic (useful when the user wants to add new contract assertions to existing code without breaking the build), and *ignore* does not evaluate the predicate at all (useful when compile-time evaluation of all contract assertions would slow the build too much; we have reports from users who find themselves in that situation).

Since the selection mechanism for the evaluation semantic is implementation-defined and the granularity of that selection is arbitrary, [P2900R14] explicitly permits compilers to use one semantic for compile-time evaluations and another for runtime evaluation, e.g., by providing two separate compiler flags. This flexibility could be useful in many scenarios; e.g., during development, we might want to temporarily disable compile-time contract checks to speed up builds while still performing runtime contract checks when testing the program.

The first proposal for C++ Contracts in Phase One, [N1613], noted that compile-time assertions can be provided with a separate feature (which later became `static_assert`) but that unifying compile-time and runtime assertions might be desirable. The next revision, [N1669], proposed to prefix compile-time assertions with `static`, thus making compile-time and runtime assertions syntactically distinct (unlike in [P2900R14]).

Phase Two Contracts did not consider compile-time assertions; the exploration of the Contracts design space [N4110] published in early Phase Three suggested that this functionality should be part of Concepts instead. The Contracts Merged Proposal [P0246R0] realized the importance of reusing the same contract assertions both at compile time and at runtime, depending on when the function in question is actually evaluated, and provided for constant evaluation of contract assertions; if the predicate were to evaluate to `false` or to not be a core constant expression, the program would be unconditionally ill-formed.

C++2a Contracts [P0542R5] added a provision that during constant evaluation, only *checked* contract assertions are evaluated; whether a contract assertion is checked is dependent on its assertion level and the build mode. [P1671R0] recognized that tying the compile-time evaluation semantic to the runtime evaluation semantic in this way is not the best design since we might want to diagnose, at compile time, contract violations that could otherwise cause UB at run time, even if the corresponding assertions are not actually checked at run time, something that [P0542R5] did not allow. The paper proposed an improvement that would have brought C++2a Contacts closer to our current design: that it is *implementation-defined* whether contract assertions that are not checked at run time are checked at compile time. However, the paper was not adopted because C++2a Contracts was removed from the C++20 Working Draft.

In the first version of the Contracts proposal in Phase Four, [P2388R0], the authors were not yet sure whether checking assertions at compile time was implementable and whether such compile-time checking should be enabled even if runtime checking is disabled. However, in the next revision, [P2388R1], the authors adopted the [P1671R0] model by specifying that if contract assertions are checked at run time, they are also checked at compile time and that it was implementation-defined whether they would be checked at compile time if they are *not* checked at run time. Note that at the time, the proposal was still assuming that whether contract assertions are checked is determined by a per-TU build mode rather than by the flexible evaluation semantics we have in [P2900R14] today.

The exact semantics of contract checks during constant evaluation, however, were not yet fleshed out very precisely. The first serious attempt at doing so was published in [P2834R1], but the paper was still assuming the existence of build modes. An update of that proposal that takes into account the flexible evaluation semantics model that had been adopted via [P2877R0] was published in [P2932R3]; around the same time, [P2894R2] was published and was dedicated to specifying the exact semantics of contract checks during constant evaluation in all the necessary detail. The proposals in these two last papers were almost identical, but [P2932R3] did not call the case where the predicate is not a core constant expression a contract violation.

The specification proposed in [P2894R2] was approved by SG21 with strong consensus.

SG21, Teleconference, 2024-01-11, Poll

For the contracts MVP, adopt the rules for contract checking during constant evaluation as proposed in P2894R2.

SF	F	N	A	SA
9	8	1	1	0

Result: Consensus

Only two aspects of this design caused discussion at the time of adoption. The first was that a predicate that is not a core constant expression is specified to be a contract violation; the case was made that it should make the program unconditionally ill-formed. However, such a specification would make the *ignore* semantic impossible because to determine whether an expression is a core constant expression, the expression must be constant-evaluated. Note that just making the contract assertion not a core constant expression *without* making it a contract violation is also impossible because that would allow SFINAE-ing on whether the contract assertion succeeds, which is in violation of the Contracts Prime Directive ([P2900R14]'s Design Principle 1).

The second, closely related question was whether the *ignore* and *observe* semantics should even exist for constant evaluation or whether a contract assertion should *always* be checked and *always* lead to a compile error. [P3079R0] made the case that insufficient justification exists for having *ignore* and *observe* during constant evaluation and that the main argument for having *ignore* — the compile-time cost of constant-evaluating all contract predicates, which might be prohibitive in some scenarios — requires more evidence. SG21 was unconvinced by that presentation, but the case made led to a split room in EWG during the first round of design review.

EWG, Tokyo, 2024-03-20, Poll 2

P2900r6: contracts in constant expressions should have enforce semantics only, and should not have ignore nor observe.

SF	F	N	A	SA
7	10	16	7	8

This question came back to SG21, which judged that the design in the Contracts proposal was correct, but stronger motivation for it was needed to convince EWG. That motivation was subsequently provided in [P3338R0], and the question was settled without further polls being taken.

Depending on how the build is configured, a contract assertion might be checked during constant evaluation in one TU and not checked in another TU; this difference can lead to an ODR violation. This situation was noted during review and described in [P2900R14] but was deemed to be an issue unlikely to cause issues in practice or to be possible to meaningfully address.

One more question that came up during wording review was if elision and duplication should indeed also be permitted during constant evaluation or if we could tighten the rules to say that, at least during constant evaluation, the predicate of a *checked* contract assertion is evaluated exactly once. We decided against making such a change because doing so would not remove the ODR issue (the contract assertion could still be checked in one TU and not checked in another), so the inconsistency between compile-time and runtime evaluation introduced by such a change seemed gratuitous.

3.6 Noteworthy Design Consequences

3.6.1 Undefined Behavior During Contract Checking

As already mentioned in Section 3.5.3, concerns were raised in [P2680R1], [P3173R0], and [P3285R0] that the evaluation of a contract assertion could be affected by undefined behavior. In Section 3.5.3, we distinguished two kinds of situations where this concern arises. The first kind, contract checks performed with the *observe* semantic being optimized out due to undefined behavior in subsequent code, is removed by introducing observable checkpoints as discussed in that section. Here, we will discuss the second kind: contract checks performed with *any* evaluation semantic (including *enforce* and *quick_enforce*) being optimized out or otherwise affected due to undefined behavior that occurs while evaluating the contract predicate of *that* check.

The following example, originally from [P2680R1] and discussed in [P2900R14], illustrates the issue:

```
int f(int a) { return a + 100; }
int g(int a) pre (f(a) > a);
```

In this program, the compiler is allowed to assume that the signed integer addition inside `f` will never overflow (because that would be undefined behavior) and to elide the precondition assertion entirely, even if the evaluation semantic is *enforce* or *quick-enforce*.

This issue is the consequence of the existing C++ rules for how expressions are evaluated and exists independently of how the assertion is written (whether with contract assertions as proposed in [P2900R14] or otherwise). However, the claim made in [P2680R1] and its follow-up papers, [P3173R0] and [P3285R0], is that because contract assertions exist to help identify and mitigate program defects, including those that could otherwise lead to undefined behavior, they themselves need special protection from being exploitable by undefined behavior.

The alternative model proposed in [P2680R1] and its follow-up papers are the so-called "strict contracts," for which the contract predicate is constrained to expressions that can be proven at compile time to be free of side effects outside their cone of evaluation (see Section 3.5.8) and free of undefined behavior — or at least certain exploitable forms of it (undefined behavior due to data races is usually excluded). The papers propose that such "strict contracts" should be the *default* and that anything else would be nonviable; [P3362R0] further claims that "strict contracts" are necessary for static analysis tools to work effectively with Contracts.

While the "strict contracts" model is certainly interesting, we decided not to adopt it for [P2900R14]. Detailed rationale for this decision can be found in [P3376R0] and [P3386R0], and we provide a brief summary below.

An important reason for this decision is that at the time of writing, "strict contracts" have not been demonstrated to be specifiable and implementable, which, to us, does not seem feasible except perhaps for a vanishingly small subset of expressions that includes certain operations on integers (see [P3499R0]). Notably, that subset does not seem to include any operations on pointers or references because we do not know how to locally prove that they point to a valid object. Further, it does not include calling any member functions of any object because we cannot, in general, prove that the `this` pointer is valid. Finally, the subset cannot include calls to *any* functions at all unless those functions themselves are marked with some kind of "strict" annotation and proven by the compiler to adhere to the same restrictions.

"Strict contracts," as we currently understand them, therefore, do not seem particularly useful for writing contract assertions in any real-world scenario. We cannot even call a trivial function like `vector.size()` inside a strict contract predicate. Further, even for those expressions that we *could* write in a strict contract predicate, such as integer addition, redefining the semantics to no longer have undefined behavior (e.g., for overflow to saturate or wrap around) would lead to the expression spuriously succeeding inside the strict predicate while still failing inside the function body; examples of such failures are given in [P3386R0].

We also disagree with the claims made in [P3362R0] regarding static analysis. We worked with five different static-analysis vendors to ensure that the Contracts proposal meets their needs (see [P3386R0]). We also believe that much of the discussion has been complicated by the word "safety" being used in different ways, causing misunderstanding of what problem the Contracts feature is actually designed to solve; see [P3376R0] and Section 2.7 in this paper. Specifically, the Contracts feature is not designed to add language guarantees to C++ to remove undefined behavior but to be *complementary* to such efforts by identifying defects during program execution that are *not* caught by such guarantees.

Overall, standardizing a subset of C++ that removes undefined behavior seems like a great idea, and interesting work is being done in this area, yet we see no reason to predicate releasing the Contracts proposal on any of that work. Instead, we should pursue approaches to remove undefined behavior independently, in a way that can be applied to the *entire* C++ language. Such approaches include static rules and restrictions — e.g., via the Profiles feature ([P3081R0]), lifetime annotations ([P2771R1]), and a new safe object model ([P3390R0]) — and the insertion of implicit lifetime checks, as proposed in [P3100R1]. Once we have those tools in place for evaluating C++ expressions in general, we can then seamlessly apply them to contract predicates as well, without the need to create a parallel language with different semantics to express those predicates.

The idea of "strict contracts" has been discussed and polled multiple times in SG21 (Contracts), SG23 (Safety and Security), and EWG over the last two years. In each of these groups, polls showed that the preferred direction is the design of [P2900R14]: contract predicates that follow the usual rules for C++ expressions, rather than some other "strict" rules that at the time of writing have not been shown to be specifiable, implementable, or applicable to real-world use cases for adding contract assertions to existing code.

The idea of "strict contracts" was first discussed in SG21 in March 2022. The first formal paper published on the subject was [P2680R0]. It was discussed in SG21 was in November 2022; additional material that was presented to SG21 but is not contained the paper is available in [P2743R0]. SG21 did not have consensus to pursue the idea further.

SG21, Kona, 2022-11-12, Poll

Given our limited resources, we encourage the author of P2680 to come back with a revision addressing the currently open questions, to be addressed in an SG21 telecon before the end of 2022.

SF	F	N	A	SA
9	5	1	3	7

Result: No consensus

Nevertheless, the SG21 chair agreed to schedule a revision of the paper in SG21. [P2700R0] summarized the many questions that SG21 had for the author of [P2680R0]. Revision [P2680R1] attempted to answer them and was discussed by SG21, who again reached no consensus to pursue this direction.

SG21, Teleconference, 2022-12-14, Poll 1

SG21 should attempt to design a model for safe programming in C++ as part of the contracts MVP, despite the existence of SG23, and considering that this would be a diversion from the current SG21 roadmap (P2695R0).

SF	F	N	A	SA
5	2	0	7	4

Result: No consensus

SG21, Teleconference, 2022-12-14, Poll 2

Notwithstanding the previous poll, the contracts MVP should contain contract-checking predicates that are free of certain types of UB and free of side effects outside of their cone of evaluation, as proposed in D2680R1, and SG21 should spend further time reviewing this design, considering that this would be a diversion from the current SG21 roadmap (P2695R0).

SF	F	N	A	SA
5	2	0	7	4

Result: No consensus

At this point, the topic of "strict contracts" was closed as far as SG21 was concerned. However, when the Contracts proposal was forwarded to EWG for the WG21 meeting in March 2024 in Tokyo, the author of [P2680R1] published [P3173R0], repeating the concerns and targeting EWG directly. EWG showed significant interest in pursuing the general concern raised.

EWG, Tokyo, 2024-03-20, Poll 8

P2900r6: contracts should expose less undefined behavior than regular C++ code does.

SF	F	N	A	SA
17	12	12	12	5

Following this EWG poll, a joint SG21/SG23 session was scheduled for the same Tokyo meeting to discuss the issue of undefined behavior in contract predicates. At this point, the joint group recognized the two kinds of issues, one that can be addressed via observable checkpoints and one that cannot (as explained above), and the two were polled separately. The group supported the design of the Contracts proposal.

SG21/SG23 Joint Session, Tokyo, 2024-03-22, Poll 1

We believe that Contracts should not be incorporated into the C++ working paper until we can specify observed contract assertions to act as optimisation barriers.

SF	F	N	A	SA
5	2	1	18	5

Result: Consensus against

SG21/SG23 Joint Session, Tokyo, 2024-03-22, Poll 2

We believe that SG21 should consider proposals for a safe programming model for C++ independently from SG23.

SF	F	N	A	SA
1	1	6	15	6

Result: Consensus against

SG21/SG23 Joint Session, Tokyo, 2024-03-22, Poll 3

We believe that Contracts should not be incorporated into the C++ working paper until a safe programming model for C++ has been specified.

SF	F	N	A	SA
3	1	1	8	16

Result: Consensus against

Following procedural complaints that the author of "strict contracts" had not been given a fair chance to present his ideas during that joint SG21/SG23 session, it was agreed that "strict contracts" should be reviewed again in SG23. A new "strict contracts" paper, [P3285R0], was presented to SG23, and no consensus was reached to pursue the idea further.

SG23, Teleconference, 2024-05-20, Poll 1

We should promise more SG23 committee time to pursuing the approach of handling certain kinds of undefined behaviour differently inside and outside of contract predicates, knowing that our time is scarce and this will leave less time for other work.

F	N	A
7	3	13

SG23 further reconfirmed that the current treatment of undefined behavior in the Contracts proposal was adequate from a safety and security perspective.

SG23, Teleconference, 2024-05-20, Poll 2

The P2900 treatment of undefined behaviour inside contract predicates is adequate for progress towards publication in an IS or TS.

SF	F	N	A	SA
18	8	0	0	3

Despite the question seemingly being settled, the topic of "strict contracts" came up yet again, ahead of the WG21 November 2024 meeting in Wrocław, with the publication of [P3362R0] and two response papers, [P3376R0] and [P3386R0], that [P3362R0] triggered. EWG discussed all three papers at the Wrocław meeting; an early draft of [P3499R0], showing the limitations of "strict contracts," was also seen at the same meeting. EWG had consensus against pursuing this direction, although sustained opposition to that decision remained.

EWG, Wrocław, 2024-11-19, Poll

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

SG21, Teleconference, 2024-06-06, Poll

D3499: 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

Following Wrocław, [P3506R0] and [P3573R0] argued again, as part of a longer list of concerns about the Contracts design, that the possibility of a contract check to exhibit undefined behavior should somehow be avoided (again without any concrete proposal regarding how this should be achieved). Another rebuttal was provided in [P3591R0]. The Contracts proposal had already, at that point, been forwarded to CWG and LWG for wording review and inclusion into the C++26 Working Paper, yet the matter was discussed again in EWG at the February 2025 WG21 meeting in Hagenberg. The group took a poll to do "something" about this concern, resulting in even stronger consensus than before against pursuing that direction.

EWG, Hagenberg, 2025-02-11, Poll

P2900: reduce the amount of Undefined Behavior in contracts.

SF	F	N	A	SA
9	7	9	22	19

Result: Consensus against

In a parallel development to "strict contracts", the idea was developed that in order to mitigate certain forms of undefined behavior, the compiler could in certain cases insert implicit runtime checks (e.g., bounds checks); if any such *implicit* contract assertion fails, this failure could be treated as a contract violation in the same way as an *explicit* contact assertion (one that the user wrote). A first version of this idea was presented in [P2811R7] but did not get consensus in SG21.

SG21, Issaquah, 2023-02-08, Poll 2

We would like to non-normatively state the recommended practice that implementations should consider treating undefined behavior in the evaluation of a contract predicate as a contract violation.

SF	F	N	A	SA
5	11	5	2	8

Result: No consensus

A much more developed version of this idea was published in [P3100R1] and was received favorably by SG21.

SG21, Wrocław, 2024-11-22, Poll 5

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

The design direction proposed in [P3100R1] and supported by SG21 is our preferred solution to the problem raised by the proponents of "strict contracts," but this proposal is intended as a post-C++26 extension, not as part of [P2900R14]. At the time of writing, [P3100R1] has not yet been reviewed by SG23 (Safety & Security) or by EWG.

3.6.2 Invalid Data Member Access in Constructors and Destructors

An issue known for a while but not deemed necessary to address until [P3172R0] is that of data members not yet being initialized when evaluating the precondition assertions of a constructor and already being destroyed when evaluating the postcondition assertions of a destructor. The paper discussed three different approaches: make accessing the values of those members undefined behavior (the default behavior if nothing else is specified and the status quo in the Contracts proposal at the time); make ill-formed using `this` in any way in those contexts; and make ill-formed calling subobject nonstatic member functions as a less restrictive version of the former option. The paper recognized that the last option is unimplementable; the second option was discussed and rejected by SG21.

SG21, Teleconference, 2024-06-06, Poll

The use of `this` either explicitly or implicitly in a precondition of a constructor or postcondition of a destructor should be ill-formed

SF	F	N	A	SA
1	3	2	7	3

Result: Consensus against

We cannot just prevent the user from using `this` in the precondition assertions of a constructor and the postcondition assertions of a destructor because important use cases exist for such assertions. While accessing the value of data members might be undefined behavior, accessing their *address*, or the address of `this` itself, is fine and useful, e.g., to assert that these objects are located in a certain region in memory.

[P3510R2] proposed a new solution to this problem: Implicit use of `this`, including naming data members directly, should be ill-formed in these contexts, and explicit use of `this`, including naming data members with `this->`, should remain well-formed. This solution allows most inadvertent accesses of data members that could lead to undefined behavior in these contexts to be rejected at compile time, and an explicit opt-in for those use cases where the address of these objects is required is available via `this->`.

This proposal achieved consensus in both SG21 and EWG and was subsequently incorporated into [P2900R14].

SG21, Wrocław, 2024-11-22, Poll 5

For P2900, make implicit access to non-static class members (both data and function) ill-formed in constructor preconditions and destructor postconditions as proposed in P3510R1.

SF	F	N	A	SA
6	7	2	0	0

Result: Consensus

EWG, Wrocław, 2024-11-22, Poll

P3510r1 Leftover properties of `this` in constructor preconditions, incorporate the paper into P2900, forward to CWG for inclusion in C++26.

SF	F	N	A	SA
16	21	4	1	2

Result: Consensus

3.6.3 Friend Declarations Inside Templates

C++2a Contracts had a rule, described in [P2900R14], designed to explicitly avoid the issues with friend declarations inside templates: If a friend declaration is the first declaration of the function in a translation unit and if that declaration has precondition or postcondition assertions, the declaration shall be a definition and shall be the only declaration of the function in that translation unit.

However, upon closer inspection, we found that this rule was not actually helpful because no way was left to enable befriending a function declared in a distinct header and remaining independent of the ordering of the class definition with the friend declaration and the separate declaration in the other header. Separately, when multiple templates need to befriend the same function, issues arise with the unpredictability of the order in which declarations will be instantiated. Rather than disallow the preconditions and postconditions on the friend declaration, allowing them to be repeated on later declarations appeared vastly more user friendly and would allow ensuring that any declaration that

might instantiate as the first declaration would have the precondition and postcondition assertions on it.

Therefore, in [P2900R14], we apply the rules of the C++ language as they are. To help the user, we introduce a definition for the term *first declaration* that is currently missing from the C++ Standard (even though the term is used in various places), and we provide a set of recommendations in [P2900R14] for writing code that avoids compiler errors due to precondition and postcondition assertions missing on the first declaration of a friend function.

3.6.4 Recursive Contract Violations

In [P2900R14], we decided against disabling contract checking during violation handling to prevent recursion and against adding any special provision to detect or mitigate recursive contract-violations; instead, if a contract violation occurs while handling another contract violation, we get a recursive call to the contract-violation handler. We chose this approach for several reasons.

First, silently disabling contract checks in code where those checks are supposed to be enabled might be surprising to the user and, more importantly, makes code evaluated during contract checking fundamentally unsafe. Second, an attempt to detect recursion by default, such as via an implicit thread-local flag, would impose overhead on *all* users and, worse, delay termination for users who need to minimize, when a violation is detected, any actions performed prior to termination.

Instead, we let the user choose whether they prefer to manually prevent recursion in their own contract-violation handler ([P2900R14] shows how to accomplish this). If the user chooses to do nothing about this issue, a recursive contract violation will typically cause a stack overflow, which should be easy enough to diagnose and fix in most cases.

The approach to preventing or handling recursive contract violations, i.e., contract violations that occur while handling another contract violation, has been changing and developing over the years. The first Contracts proposal in Phase One, [N1613], proposed that during contract-violation handling, contract assertions are not checked, thus preventing recursion. In revision [N1773], this rule was removed with no motivation given, and it was added back again in revision [N1866].

In Phases Two and Three, recursive contract violations were allowed with no special treatment given; [P0247R0] states that this change does not appear to cause any problems.

The first discussion of the issue in Phase Four can be found in [P2388R4]. Whether to disable contract checking during violation handling to prevent recursion is listed in the paper as an open issue. A more thorough and up-to-date discussion can be found in [P2811R7], the paper that was adopted by SG21.

3.6.5 Concurrent Contract Violations

The issue of the contract-violation handler potentially being invoked multiple times concurrently, leading to data races if the handler has not been designed in a thread-safe way, was raised during design review. [P2900R14] explains why we do not explicitly prevent such races and leave the issue up to the user.

3.6.6 Throwing Violation Handlers

Another design aspect of [P2900R14] that caused considerable discussion is that the user might make the user-defined contract-violation handler `noexcept(false)` and then throw an exception from that handler. [P2900R14] specifies that in this case, the exception behaves as if it was thrown from the function body, and stack unwinding happens as normal until the exception either reaches a `catch` clause or hits a `noexcept` barrier, which causes `std::terminate` to be called. In particular, if no `noexcept` barrier is encountered, throwing from the contract-violation handler allows the user to prevent termination that would otherwise happen if the contract assertion that failed was evaluated with the *enforce* semantic.

This property is important because for some applications, termination is completely unacceptable (see [P2698R0]), yet continuing into code past a contract-violation is likely to cause a crash and is also unacceptable. Throwing an exception and unwinding the stack is the only other portable option C++ provides.[22] At the same time, for other applications, allowing the program with a detected bug to continue in any form is also completely unacceptable (see discussion in [P2388R0]).

Therefore, [P2900R14] offers all options. If a use case requires it, we can throw an exception from the contract-violation handler and catch that exception further up the stack in an attempt to recover and fall back to some sane state that allows the program to continue despite a bug. This strategy may not be appropriate for applications with stringent security needs (e.g., fail fast as soon as the program is found to be in any kind of invalid state) and requires meticulously following the Lakos Rule, i.e., not annotating functions having preconditions with `noexcept`, even if they cannot throw when called in contract; otherwise, an exception thrown from the contract-violation handler would cause the very termination that this strategy seeks to avoid. This use case requires the entire codebase to be prepared in a particular way. Nevertheless, the use case is important to support even though we expect it to be less common. The more usual use case is that a contract violation with the *enforce* semantic should cause program termination. In this case, neither following the Lakos Rule nor special preparation of the codebase is necessary; the desired effect can be achieved by *not* throwing an exception from a user-defined contract-violation handler.

The importance of allowing throwing contract-violation handlers is discussed in more detail in [P3318R0]. Additional discussion can be found in [P2339R0], [P2811R7], [P2831R0], and references therein.

One key specification aspect — and a noteworthy difference between the contract-violation handler on one hand and other replaceable functions, such as global `operator new`, on the other — is that the implementation provides a *definition* for the contract-violation handler but no *declaration*. This technique allows the implementation to provide a default contract-violation handler that is `noexcept(true)` and the user to replace it with a user-defined contract-violation handler that is `noexcept(false)` (or vice versa). Because no declaration for `::handle_contract_violation` is provided in any Standard Library header, when a user declares and defines their own handler, they are declaring it for the first time and thus are free to declare it with the `noexcept`-ness of their choice.

[22] An explanation of why other options, such as `longjmp`-ing out of the function or raising a signal, are not viable options can be found in [P2831R0]. That paper has a different context (i.e., negative testing), but its analysis of nonterminating contract-violation handling strategies applies equally to running code in production.

WG21 members raised a concern that throwing from a contract-violation handler creates a new code path that could cause exceptions to be thrown in libraries and other contexts that were never designed with exception handling in mind. However, because we expect throwing contract-violation handlers to be a relatively rare and specialized use case, we don't expect that all C++ code should support this strategy. A library vendor can choose to document that they do not support it.

Detecting at compile time whether a contract-violation handler could throw an exception is not possible — at least not without new inventions in the ABI — because this is a link-time decision (see Section 3.5.9). Therefore, we do not propose such functionality in [P2900R14].

The question of throwing exceptions from the contract-violation handler is linked to the question of how contract assertions should interact with the `noexcept` operator.

Due to the Contracts Prime Directive ([P2900R14]'s Design Principle 1) and since exceptions thrown from a contract-violation handler triggered by a `pre` or `post` behave like exceptions thrown from the body of the affected function, a `pre` or `post` never changes the exception specification of a function. (Throwing from a contract-violation handler triggered by a `pre` or `post` on a function declared `noexcept` will hit that `noexcept` boundary and call `std::terminate`.)

However, we would have a problem if *deducing* the exception specification of a function or of an expression were possible, in a context where the presence of a contract assertion could affect the result, i.e., if we had to determine whether a contract assertion *itself* is potentially throwing (i.e., what its `noexcept`-ness is). On one hand, we cannot specify that contract assertions are `noexcept(false)`, because then the mere presence of a contract assertion could change the result of the `noexcept` operator and thus violate the Contracts Prime Directive; on the other hand, we cannot specify that contract assertions are `noexcept(true)` because they can in fact throw an exception if the user installs a throwing contract-violation handler, and whether that is the case is unknowable at compile time.

What the correct answer should be is fairly contentious. (See discussion in [P2969R0], Section 3.7 of [P2932R2], [P3113R0], and [P3114R0], which collectively reflect the original discussion in SG21, and in [P3541R1], which is a more recent take on this question.) Therefore, [P2900R14] is designed such that the user *cannot* ask the question of whether a contract assertion itself is potentially throwing. This design goal is why `contract_assert` is a statement, not an expression (see Section 3.2.2): to avoid having to specify the value of `noexcept(contract_assert(x))`. This goal is also part of the reason why functions with deduced exception specifications, in particular implicitly defined defaulted special member functions, cannot have precondition or postcondition assertions (see Section 3.3.3).

All C++ Contracts proposals in Phase One and Phase Two allowed contract-violation handlers to throw an exception; how contract assertions should interact with the `noexcept` operator was not considered during that time. This design aspect was first investigated properly at the beginning of Phase Three. [N4110] reconfirmed that throwing contract-violation handlers should be possible but stipulated that such an exception should escape from a function even if it was `noexcept` since contract-violation handling should happen caller-side. [N4248] contains the first in-depth exploration of the interaction of contract assertions with `noexcept`; the paper lists various options and their tradeoffs. [P0147R0] explained that adding a contract assertion to existing code should *not* change its `noexcept` properties; [N4293] and [P0166R0] showed that throwing from a contract-violation handler is fundamentally incompatible with `noexcept`.

[P0246R0] specified that if an exception thrown from a contract-violation handler encounters a `noexcept`, the program is terminated; this specification has remained in place since and was reconfirmed multiple times in subsequent Contracts proposals.

In Phase Four, what contract-violation handlers should or should not be allowed to do, including whether they should be allowed to throw, was explored in [P2339R0]. Throwing from the contract-violation handler was further explored in [P2784R0]; the paper proposed an alternative to throwing, in the form of so-called "abortable components."

Paper [P2828R0] highlighted the concrete design issues regarding the interaction between contract assertions and `noexcept` and the need to establish concrete semantics. Such concrete semantics were proposed in [P2811R7]. These papers were discussed in SG21 and led to the following design decisions that now form part of [P2900R14].

SG21, Teleconference, 2023-05-18, Poll 1

Whether a contract-checking annotation is evaluated should never change the result of the `noexcept` operator or the type of a function pointer to which the contract annotation is attached, provided that in both cases the code is well-formed.

SF	F	N	A	SA
13	5	1	0	0

Result: Consensus

SG21, Teleconference, 2023-05-18, Poll 3

Throwing an exception from a contract violation handler shall invoke the usual exception semantics: stack unwinding occurs, and if a `noexcept` barrier is encountered during unwinding, `std::terminate` is called, as proposed in P2811.

SF	F	N	A	SA
10	7	2	0	0

Result: Consensus

SG21, Teleconference, 2023-05-18, Poll 4

Preconditions, postconditions, and assertions shall all behave the same way with respect to throwing an exception from the predicate or the violation handler.

SF	F	N	A	SA
10	8	0	0	0

Result: Consensus

With the above design decisions, the design that we have in [P2900R14] was finalized. However, during design review in EWG, the concerns over the possibility to throw from the contract-violation handler were raised again, leading to a split room on the issue.

SG21, Teleconference, 2024-06-06, Poll

P2900r6: contracts should not allow throwing exceptions out of a violation handler.

SF	F	N	A	SA
8	16	12	7	14

Thus, the question was referred back to SG21. After more discussion, SG21 decided that the current design was correct, but more motivation was needed. This decision led to the publication of [P3318R0] and another discussion in EWG, at the end of which the room was sufficiently convinced that throwing contract-violation handlers should remain in the paper.

EWG, St. Louis, 2024-06-24, Poll

P2900r7: Contracts should not allow throwing exceptions out of a violation handler.

SF	F	N	A	SA
0	5	4	9	16

Result: Consensus against

The topic of throwing violation handlers came up again during LEWG design review of [P2900R10] at the WG21 meeting in November 2024 in Wrocław. A major compiler vendor raised a concern that since the compiler cannot determine at compile time whether `::handle_contract_violation` is potentially throwing, it must assume that the function might throw. In turn, this assumption has an impact on code generation. Specifically, when calling a potentially throwing function from a `noexcept` function, the compiler must ensure that no exception can actually escape. Various implementation strategies exist for this task,[23] some of which lead to code bloat, and the specific concern was that such code bloat would be forced onto the compiler. However, the authors found a straightforward, working implementation strategy that avoids this problem.[24] The concern was, therefore, withdrawn.

[23] Clang's strategy is to generate code akin to

```
try { f(); } catch (...) { std::terminate(); }
```

which can lead to code bloat when calling a potentially throwing function from a `noexcept` function. GCC's strategy is to generate a special kind of frame in the exception-handling tables so that the stack unwinding process will terminate when it encounters that frame. This solution leads to slightly larger exception tables, but the "happy case" code path is optimal.

[24] Instead of calling the potentially throwing contract-violation handler, the compiler can generate a dummy function exactly once in the program, which effectively does the following:

```
void __call_contract_violation_handler_from_noexcept(...) noexcept {
  try {
    ::handle_contract_violation(...);
  } catch (...) {
    std::terminate();
  }
}
```

Then, when failing a contract check from a `noexcept` function, the compiler can call that function instead. When failing a contract check from a non-`noexcept` function, `::handle_contract_violation` can be called instead, and no issue occurs. This solution works, is straightforward, and solves the code bloat issue on Clang.

Throwing violation handlers came up one more time in the context of [P3541R1], which asked how contract-violation handlers should interact with `noexcept` in cases where contract violations can occur in more places than proposed by [P2900R14] and, in particular, during evaluation of built-in *core language* expressions, such as pointer dereference (a future extension proposed in [P3100R1] and [P3081R1]). One of the proposed solutions was again to remove the ability of contract-violation handlers to throw an exception, and SG21 again had consensus against such a solution.

SG21, Teleconference, 2025-01-09, Poll 2

If P3081 Profiles add implicit contract checks to core language expressions such as pointer dereference or array indexing, we want to require that contract-violation handlers must be `noexcept` (P3541R1 Option 2).

SF	F	N	A	SA
0	1	2	9	7

Result: Consensus against

Throwing violation handlers were discussed one last time before [P2900R14] was accepted into the C++26 Working Paper. To address concerns that throwing contract-violation handlers create a new and potentially unintended codepath, [P3577R0] proposed to introduce a normative requirement to the Contracts paper that the *default* contract-violation handler shall not exit via an exception. Thus, the new code path will not be introduced unless the user has *explicitly* opted into it by replacing the default contract-violation handler with a user-provided one. That paper was written in the context of introducing *implicit* contract assertions (as proposed in [P3081R2], [P3100R0], [P3229R0], and [P3599R0]), which are not part of [P2900R14] but are intended as a future extension, but the proposed normative requirement would apply to [P2900R14] as well. Introducing this requirement was supported by SG21.

SG21, Teleconference, 2025-01-23, Poll 1

Forward P3577R0 “Require a non-throwing default contract-violation handler” to EWG for C++26.

SF	F	N	A	SA
4	11	3	1	2

Result: Consensus

However, EWG did not follow the recommendation from SG21 guidance and did not accept this proposal.

EWG, Hagenberg, 2025-02-11, Poll

P3577r0 Require a non-throwing default contract-violation handler: forward to CWG for inclusion in C++26.

SF	F	N	A	SA
7	13	15	2	3

Result: No consensus

In [P2900R14], as a result of EWG's decision, a default contract-violation handler that exits via an exception is conforming, even though it goes against the recommended practice and we do not expect any major compiler to ever provide such a default.

3.6.7 Differences Between Contract Assertions and the `assert` Macro

Earlier Contracts proposals, in particular those in Phase Two (Macro Contracts), evolved entirely from macro-based solutions and brought with them all the benefits and problems of building a contract-checking facility on the back of the preprocessor. After the macro contracts proposal failed in plenary and focus turned toward language-based solutions, we clearly saw that the problems of macros could be fixed instead of migrated into matching problems with a language-based facility.

- Anyone who has ever supported a contract-checking facility based on macros has experienced bit rot, where the predicates of assertions no longer compile and thus prevent enabling exceptions without paying off silently accrued technical debt.
- Being completely explicit about the code transformation applied to contract assertions, as a preprocessor macro is, limits the behavior of a contract assertion to exactly those things that could be accomplished by regular C++ code. Such limitations would prevent even considering aspects of the Contracts feature, such as `const`-ification, strict contracts, elision, or duplication.
- Using the preprocessor for control of the evaluation semantic means generating different tokens for different semantics, which is inevitably an ODR violation. This problem with using macros for contract assertions is a fundamental one that only switching to a language-based solution can fix.
- Using the preprocessor with a known control macro, such as `NDEBUG`, allows arbitrary additional ABI-changing declarations to be wrapped in the same macro. While this feature is useful, it brings the same ODR violations that motivated switching away from macro-based solutions in the first place. A future feature to allow state, in addition to code, to be kept as part of larger contract-assertion evaluations, such as the `contract_support` statements described in Section 3.4.4 of [P2755R1], will be needed to facilitate such use cases.

One other difference between `contract_assert` and `assert` is that the latter can be used in expressions, while `contract_assert` is only a statement. This difference is a feature of `assert` that was not included in C++2a Contracts and one many participants in the Standardization process were unaware of. Allowing `contract_assert` to be an expression, however, would require answering whether it is a potentially throwing expression; see Sections 3.2.2 and 3.6.6 for why determining an answer to that question is problematic.

3.7 Standard Library API

Handling contract violations via a user-provided callback is an established, well-tested approach that is deployed in many modern assertion facilities. In [P2900R14], this callback is the *user-defined contract-violation handler* (see Section 3.5.9). To provide information about the contract violation to that handler, a relatively small amount of library API is necessary. Providing a *standard* API for this purpose is a significant step forward compared to existing nonstandard assertion facilities since it allows the user to implement, for the entire program, a single contract-violation handler that can

handle, in a uniform way, a wide variety of contract violations originating from any component of that program.

The bulk of that Standard Library API, as proposed in [P2900R14], originates from [P2811R7], the primary contract-violation handling proposal that was approved by SG21 and subsequently added to the Contracts paper. A detailed motivation and discussion of this API can be found in that paper. Here, we provide a summary of that motivation as well as the history of this library API, which includes several subsequent papers that proposed modifications to the design described in [P2811R7], considered by SG21, and, in some cases, approved for the Contracts paper.

3.7.1 The `<contracts>` Header

The proposed library API has a very specific purpose — enabling the implementation of user-defined contract-violation handlers — and is, therefore, located in its own header. Further, all the contents of that header are declared in a separate namespace, `std::contracts`, rather than in namespace `std`. Though such nested namespaces are uncommonly used in the Standard Library, this case presented good reasons for doing so.

- Users of the types and functions provided are going to be relatively few and far between. In general, for any given application, at most one contract-violation handler will be implemented; we expect a common violation handler to be typically provided by application frameworks used across an entire enterprise. Therefore, the length of the qualified name (and of any names introduced in this proposal) is not a consideration.
- Because the names introduced here are used so infrequently, they must not inadvertently pollute the set of names visible for daily use of the language.
- If names were to be put directly in `std`, they would all need to contain something to clearly mark them as part of the Contracts feature, such as a `contracts_` prefix in front of every name in the proposed API. Though this prefix might be viable for the names proposed in [P2900R14], all planned future extensions beyond [P2900R14] (see Section 2.3) would be similarly bound by that need, permanently tying us to such a prefix. This prenamespacing approach to naming seems ill advised. That said, one of the names proposed in [P2900R14] indeed has a `contract_` prefix: `contract_violation`. Motivation for this naming choice can be found in Section 3.7.3.

In the past, Contracts proposals for C++ have vacillated on this question. The first Contracts proposal that introduced some kind of library API, [N1773], placed all proposed names in namespace `std` (e.g., `std::class_invariant_broken()`). This strategy was retained throughout Phase One. In Phase Two, proposals switched to the nested namespace strategy: [N3604] proposed a header, `<preassert>`, and a namespace, `std::precondition`, and [N3818], which expanded the proposal to assertions other than just for preconditions, changed this to header `<contract_assert>` and namespace `std::contract`. [N3963] switched back to placing all names into namespace `std`, which was retained throughout Phase Three as well: C++2a Contracts [P0542R5] proposed `std::contract_violation`.

In Phase Four, the nested namespace approach was reintroduced by [P2811R5], which originally proposed header `<contract>` but namespace `std::contracts`. The singular form was chosen so that the header name would be consistent with existing headers, such as `<exception>` or `<coroutine>`. However, SG21 recognized that the header should instead be consistent with the namespace it

introduces, for which we have existing practice with header `<ranges>` and namespace `std::ranges`. The proposal was accepted into the Contracts paper with that modification.

SG21, Varna, 2023-06-15, Poll 1

The `<contract>` header proposed in P2811R5 should be renamed to `<contracts>` to be consistent with the proposed namespace contracts.

SF	F	N	A	SA
10	8	2	1	0

Result: Consensus

In addition, SG21 decided to make the `<contracts>` header freestanding since no reason exists to preclude using contract-violation handling on a freestanding implementation and since existing obstacles to making the header freestanding had been removed in the meantime (in particular, by making `std::abort` freestanding; see Section 3.5.4).

SG21, Teleconference, 2024-02-22, Poll

The contents of header `<contracts>` should be freestanding.

SF	F	N	A	SA
8	6	1	0	0

Result: Consensus

During a first round of LEWG design review at the March 2024 WG21 meeting in Tokyo, the suggestion was raised again to use a prefix `contract_` instead of a nested namespace. However, this suggestion was rejected by SG21.

SG21, Tokyo, 2024-03-21, Poll

The library facilities proposed in P2900R6 should be in namespace `std` rather than in a nested namespace `std::contracts`.

SF	F	N	A	SA
1	6	4	9	2

Result: No consensus

3.7.2 Enumerations

[P2900R14] introduces three enumerations: `assertion_kind`, `detection_mode`, and `evaluation_semantic`. The motivation for the existence and functionality provided by these enumerations is provided in [P2811R7] and in [P2900R14] itself. Here, we provide additional motivation and history for the chosen names as well as a few other aspects of the proposed enumerations.

In general, the enumerators in all enumerations are kept short due to the use of `enum class` that avoids polluting the namespace in which they reside or conflicting with overly general terms. However, the names of the enumerations themselves are chosen to be more descriptive.

The enumeration `assertion_kind` was originally called `contract_kind` in [P2811R7]. A first round of LEWG design review at the March 2024 WG21 meeting in Tokyo noted the observation that the duplication of the word "contract" in the fully qualified name `std::contracts::contract_kind` was somewhat awkward, and removing either the nested namespace or the `contract_` prefix to make the fully qualified name either `std::contracts::kind` or `std::contract_kind` was suggested. SG21 did not achieve consensus for either approach since good motivation exists for both a nested namespace (see above) and a sufficiently descriptive name. However, SG21 decided to remove the duplication by choosing the even more descriptive and precise name `assertion_kind` since the enumeration describes specifically the syntactic form of the violated contract assertion, not the kind of (plain-language) contract.

SG21, Tokyo, 2024-03-21, Poll

In P2900R6, rename enum `contract_kind` to `assertion_kind`.

SF	F	N	A	SA
1	14	8	1	1

Result: Consensus

The enumerators in `assertion_kind` name the token that is used to introduce the contract assertion that was violated when it was a contract assertion that detected a violation. For assertion statements introduced with `contract_assert`, the `assert` enumerator is used instead because `contract_assert` is a full keyword and therefore not available as an enumerator name. Note that this enumerator is fine to use even in the presence of `<cassert>` due to the `assert()` macro being a function-like macro. The `assert` enumerator will never be used prior to an opening parenthesis, and thus the function-like macro will not be applied to uses of the enumerator.

The enumeration `detection_mode` could arguably be called `contract_violation_detection_mode`, but that level of verbosity seems unnecessary. Having one enumerator omit the common prefix prevents setting the same precedent we aimed to avoid by using a namespace; i.e., we wanted to avoid requiring all future Contracts-related additions to have a prefix of `contract_`.

The enumeration `detection_mode` has two enumerators: `predicate_false` and `evaluation_exception`. The second enumerator is deliberately not called `predicate_exception` because various possible future extensions (core-language preconditions, procedural interfaces, and so on) exist where the code being evaluated is not actually a boolean predicate but still might potentially exit erroneously via an exception when attempting to identify a contract violation.

As originally proposed in [P2811R7], the enumeration `detection_mode` had a third enumerator, `evaluation_undefined_behavior`, to specify a condition in which the detected bug is not a contract violation in the [P2900R14] sense, but rather an instance of undefined behavior that the implementation has the means to detect at run time. Having a consistent way to funnel this kind of condition to the contract-violation handler and to handle it intelligently there is a powerful approach to mitigate undefined behavior in C++. However, the presence of `evaluation_undefined_behavior` in the Contracts paper raised doubts about whether it was indeed the correct API for this approach and whether such an API should be included in a first version of a C++ Contracts facility. These concerns are described in detail in [P3073R0]. As proposed in that paper, `evaluation_undefined_behavior` was removed from the Contracts proposal.

SG21, Teleconference, 2024-02-08, Poll 1

Remove the enum value `detection_mode::evaluation_undefined_behavior` from the Contracts MVP, as proposed in P3073R0.

SF	F	N	A	SA
7	5	3	2	0

Result: Consensus

Following this decision, [P3100R1] was developed, proposing a much more comprehensive approach to detect undefined behavior at run time and funnel it into the contract-violation handling mechanism. This proposal is currently being pursued as a future extension.

Another proposed modification of `detection_mode` was the proposal in [P3102R0] to add an enumerator, `predicate_would_be_false`, distinct from the enumerator `predicate_false`, to distinguish a contract violation in which the predicate was evaluated and its value was found to be `false` from a contract violation in which that determination was made without evaluating the predicate (see Section 3.5.7). SG21 rejected this proposal since we currently have insufficient implementation and deployment experience with compilers that can actually make that distinction.

SG21, Teleconference, 2024-02-08, Poll 2

Add the enum value `detection_mode::predicate_would_be_false` to the Contracts MVP, as proposed in P3102R0.

SF	F	N	A	SA
2	0	6	3	3

Result: No consensus

The enumeration `evaluation_semantic` was originally called `contract_semantic` in [P2811R7]. As for `assertion_kind` and following LEWG feedback that duplication of the word "contract" is not desired, the duplication was resolved by choosing a more descriptive and precise name.

SG21, Tokyo, 2024-03-21, Poll

In P2900R6, rename enum `contract_semantic` to `evaluation_semantic`.

SF	F	N	A	SA
2	16	7	0	0

Result: Consensus

[P3237R2] proposed to specify semantics conceptually in terms of their properties (e.g., checking or non-checking, terminating or nonterminating, calling the contract-violation handler or not calling it) rather than enumerating them, as is done in [P2900R12], and then to either remove the enumeration entirely or to retain it but to use the enumerators as aliases for certain property combinations. However, SG21 rejected this approach.

SG21, Teleconference, 2024-10-24, Poll 2

We are interested in specifying evaluation semantics conceptually in terms of properties as presented in P3237R2 "Matrix Representation of Contract Semantics".

SF	F	N	A	SA
1	1	9	7	1

Result: Consensus against

SG21, Teleconference, 2024-10-24, Poll 4

We are interested in expressing contract semantics in code in terms of semantic aliases.

SF	F	N	A	SA
1	0	3	10	1

Result: Consensus against

The reasons for not pursuing this direction are discussed in more detail in Section 3.3 of [P3227R1]. One such reason is that the most common operation in a contract-violation handler is *logging*, which is much easier to do with an enumeration value than with a collection of properties on which we must branch. Another is that using named semantics rather than properties of the contract-violation handling is an important improvement of [P2900R14] over earlier proposals, as discussed in detail in Section 3.5.4, and this design should be directly reflected in the library API as well by treating the named semantics as *primary* entities.

Note that `evaluation_semantic` provides enumeration values for all four standard semantics, not just the two that call the contract-violation handler. That is, enumeration values are available for `ignore` and `quick_observe` even though `contract_violation::semantic()` will never return those values. The motivation for their inclusion is that `evaluation_semantic` is also intended as a vocabulary type to refer to evaluation semantics in other contexts. No such context besides the contract-violation handler currently exists in [P2900R14]. However, compiler vendors may provide, as an extension, labels on the assertion to specify the desired semantic explicitly. (The Clang implementation already provides such an extension.) The `evaluation_semantic` is helpful for this purpose. Not having to switch to a different enumeration seems preferable if such labels are upgraded from vendor extensions to Standard features in a later standardization cycle. (See [P3400R0] for a proposal.)

Note further that [P2900R14] does not associate the numerical values of any enumerators with particular properties, such as associating a bit of the value of `evaluation_semantic` with whether that semantic is a checking semantic or a terminating semantic. Instead, the values are simply `1`, `2`, and so on in no particular order. Such properties should instead be queried via functions like `is_terminating()` (see below), not via the enumerator value. Detailed rationale for this design is provided in Section 3.4 of [P3227R1]. For the reasons provided in that paper, the proposal in [P3237R2] to use the different bits of the `evaluation_semantic` value to stand for different properties of the semantic in question was rejected by SG21.

SG21, Teleconference, 2024-10-24, Poll 3

We are interested in redefining enum `evaluation_semantic` in terms of a bit mask.

SF	F	N	A	SA
1	0	2	8	6

Result: Consensus against

Further, no enumeration value is defined to be `0`. While the strangeness of one of these enumerations having no valid value, if zero-initialized, has been suggested, we see the usefulness in being able to detect the case in which the enumeration has not been explicitly initialized with a valid value; the proposed design allows us to do this.

Finally, the underlying type of all enumerations was originally specified to be `int` in [P2811R7]. SG21 later changed this type to be unspecified to address LEWG feedback that `int` is overly specific in this case and that the only important requirement is that the type fits all possible values, including vendor-provided ones.

SG21, Tokyo, 2024-03-21, Poll

For the enums in P2900R6, make the underlying type unspecified; mention in the front matter of P2900 that the design intent is that the underlying type needs to be large enough to hold all possible values, including any vendor-provided ones.

SF	F	N	A	SA
2	14	5	2	0

Result: Consensus

3.7.3 The Class `std::contracts::contract_violation`

The class `std::contracts::contract_violation` is the centerpiece of the proposed library API. Through this class, the implementation provides information about the contract violation to the contract-violation handler. (See Section 3.5.9 for the underlying mechanics of how this object is created and passed in.)

The API of this class consists entirely of nonvirtual `const` member functions that return values. No data members for this class are prescribed by [P2900R14] since we do not wish to impose, on implementations, any particular constraints on the ABI of this object; extending it without an ABI break should be easy ([P2900R14]'s Design Principle 16, No ABI Break).

Regarding naming rationale, the type `std::contracts::contract_violation` is so named because it encapsulates the information describing a particular check of a contract that failed to verify compliance with the corresponding contract. The prefix `contract` is present to leave room in the future for other contract-related things that might be violated. A more specific prefix, such as `assertion`, is not used exactly to avoid making this prefix too specific to prevent invoking the contract-violation handler when contracts are violated but the violation is identified through a mechanism that is not necessarily a contract assertion.

The suffix `violation` is again intended to capture the general property of what happened — a contract was violated — without limiting the type to representing only cases in which a contract assertion has failed or to implying that a plain-language contract as written was explicitly breached.

The member functions of `contract_violation`, like the enumerators in the various enumerations, are significantly shorter with no need to disambiguate. Thus, the `kind` and `semantic` member functions need no further qualifications.

The `comment` member function is expected to contain the textual representation of the expression in the contract violation when a contract assertion is violated. For builds that choose to elide such information from the distributed executable, the value of `comment` might instead be text that explains that such elision has happened (`"no data available"`). For future contract `kinds` that do not necessarily have an expression, this field is open-ended in its name to allow for flexibility in what to provide. For these reasons, we considered but ultimately rejected naming options like `expression`, `code`, `source_code`, or `source_text` that might better capture the most common use of this property but would make other expected uses inconsistent.

Providing a library API to query properties of the violated contract assertion in the contract-violation handler has a long history. Phase One proposals initially did not provide such an API. [N1773] contains the first API sketch for a `contract_violation` object (called `assertion_context` at the time), even though it was not yet being proposed.

The first proper proposal for such an API can be found in [N3604] in Phase Two: In this paper, the contract-violation handler takes four parameters (three `const char*` parameters, for the expression text, file name, and line number, respectively, and one parameter for the contract level). [N3753], which revised [N3604], updated the interface to the contract-violation handler to consolidate the passed-in information into a single struct, `std::precondition::assert_info`, rather than many individual parameters. This choice was made in response to LEWG feedback to enable easy addition of further information to that struct in future Standards. [N3963] later changed the name of the struct to `std::out_of_contract_info`. The next revision, [N3997], changed it to `std::contract_violation_info`.

In Phase Three, [P0246R0] proposed an API in which the object passed into the contract-violation handler was a `std::source_location` designating the source location of the violated assertion. [P0380R0], the initial merged proposal that later led to C++2a Contracts, specified a violation handler that took an argument of type `std::violation_info`, which had four data members (three `const char*` members, for the expression text, file name, and function name, respectively, and an `int` for the line number; note that the source location was again expressed via individual strings or numbers, not through a `std::source_location` object, to remove any dependencies on the then-upcoming `source_location` proposal which had not yet successfully been moved from the Library Fundamentals V3 by [P1208R6]). [P0542R0], which contained the wording drafted for [P0380R1] that was to be merged into the C++20 Working Paper, changed the name of the contract-violation object to `std::contract_violation` and notably also changed the four data members to four functions (`comment()`, `file_name()`, `function_name()`, and `line_number()`, respectively). A later revision, [P0542R1], added a fifth member function, `assertion_level()`. Another revision, [P0542R2], changed the various string-like properties from `const char*` to `std::string_view`. That revision also clarified that for a precondition violation, the source location from the call site may be returned instead of that of the actual precondition assertion. A later paper, [P1639R0], would have

reintroduced `source_location` and removed `string_view` if C++2a Contracts had not been removed from the draft. In this form, the library API was retained until the end of C++2a Contracts.

In Phase Four, following usage experience with an implementation of [P1607R1], the `semantic()` property was added. Taking the guidance from [P1639R0] provided by LEWG in the C++20 cycle, which we assume would have been adopted if Contracts had not been removed, string-like properties have been changed back to `const char*` and no longer use `std::string_view`, and the source location is again expressed via `std::source_location` rather than via individual properties for the line number, file name, and function name.

SG21 discussion has revealed a desire to have a forward-compatible way to identify a continuing evaluation semantic (one where the program will *not* be terminated after the contract-violation handler returns) when a violation handler has been invoked, so the `will_continue()` member function has been added to the `contract_violation` object. Incorporating much earlier proposals' (such as [N1669]'s) abilities to distinguish preconditions, postconditions, and assertions when handling a violation, the `kind()` member function has been added.

Finally, this entire library API was accepted into the Contracts paper (which initially did not have contract-violation handlers) via [P2811R7]. Alongside [P2811R7], SG21 also considered [P2853R0], which contained a number of proposals to change several important aspects of the API. However, none of the proposals in [P2853R0] were accepted by SG21 because the rationale was found to be unconvincing.

SG21, Varna, 2023-06-15, Poll 3

In the Contracts MVP, `handle_contract_violation` should return a value of type `std::contract_resolution` rather than void.

SF	F	N	A	SA
0	1	3	8	10

Result: Consensus against

SG21, Varna, 2023-06-15, Poll 4

In the Contracts MVP, should `std::contract_violation` be copy and move constructible?

SF	F	N	A	SA
1	0	2	10	8

Result: Consensus against

SG21, Varna, 2023-06-15, Poll 5

Remove enum class `contract_semantic` and `std::contract_violation` member function `semantic()` from the Contracts MVP.

SF	F	N	A	SA
1	3	9	3	3

Result: No consensus

SG21, Varna, 2023-06-15, Poll 6

We should normatively require `std::contract_violation::comment()` to return either the source code text of the predicate expression or an empty string

SF	F	N	A	SA
0	2	5	7	9

Result: Consensus against

[P3073R0] found that the specification of the member function `will_continue()` was broken. Because no consensus was achieved on the correct specification strategy for this feature, `will_continue()` was instead removed from the Contracts paper.

SG21, Teleconference, 2024-02-08, Poll 3

Remove the function `contract_violation::will_continue()` from the Contracts MVP, as proposed in P3073R0.

SF	F	N	A	SA
3	12	4	0	0

Result: Consensus

The functionality was added again later via the new member function `is_terminating()` with an updated specification that does not suffer from the problems of the earlier version. The last addition to the library API proposed in [P2900R14] was `evaluation_exception()`. Both member functions were added via [P3227R1].

SG21, Teleconference, 2024-10-24, Poll 1

Apply the changes proposed in P3227R1 "Fixing the library API for contract violation handling" as presented to P2900.

SF	F	N	A	SA
6	14	0	0	0

Result: Consensus

[P3227R1] provides detailed rationale for why these two additional member functions are necessary and why the desired functionality cannot be accomplished through other means, such as by querying `semantic()` or `detection_mode()`. Part of the problem is that both return *extensible* enumerations, which means that portably writing an exhaustive switch is impossible.

During a first round of LEWG design review of the proposed Contacts library API at the March 2024 WG21 meeting in Tokyo, the duplication of the word "contract" in the fully qualified name `std::contracts::contract_violation` was found to be somewhat awkward, and shortening it to `violation` was proposed. However, SG21 rejected this suggestion. (See further above in this section for naming rationale that motivated this decision.)

SG21, Teleconference, 2024-04-04, Poll 6

For the Contracts MVP, rename `contract_violation` to `violation`.

SF	F	N	A	SA
1	1	4	4	4

Result: Consensus against

A quirk of the library API proposed in [P2900R14] is that a `std::exception_ptr` to an exception that escaped the evaluation of a contract predicate can be accessed in the contract-violation handler not only by calling the member function `evaluation_exception`, but also by calling the pre-existing Standard Library function `std::current_exception()`. However, using `std::current_exception()` for this purpose can be error prone, because, unlike `evaluation_exception()`, which will return a nonempty `std::exception_ptr` only if the exception was thrown during predicate evaluation, `std::current_exception()` returns a pointer to the currently handled exception regardless of whether it was thrown during predicate evaluation or somewhere else in the program (i.e., the predicate evaluated to `false`, but the contract violation occurred inside an unrelated `catch` clause). In other words, `evaluation_exception()` and `std::current_exception()` may or may not point to the same exception, depending on the state of the program. The possibility to remove this footgun by separating the handling of exceptions during contract checking from the handling of exceptions elsewhere in the program was explored in detail in [P3417R1]. The paper concluded that the tradeoffs of such a design change are unfavorable, and the design of [P2900R14] in that area should, therefore, remain as is.

3.7.4 The Function `invoke_default_contract_violation_handler`

The function `invoke_default_contract_violation_handler` is a function with the same behavior as the default contract-violation handler specified by the Standard. It is named in such a way for two reasons.

1. It takes a verb form because, in general, free functions in the Standard Library have verb forms.
2. It is *not* the default contract-violation handler itself, which is not directly callable by the user[25] but is instead a function with the same behavior.

During a first round of LEWG design review at the March 2024 WG21 meeting in Tokyo, the name `invoke_default_contract_violation_handler` was observed to be very long, and suggestions for shorter names were given. SG21 rejected all such suggestions and preferred to retain the original name, `invoke_default_contract_violation_handler`, for the reasons stated above.

[25] The Standard Library provides only a definition, not a declaration, for the default contract-violation handler (see Section 3.5.9); therefore, the user cannot name it directly. If a user defines their own contract-violation handler, the name `std::contracts::handle_contract_violation` will instead refer to that user-defined handler. If they do not define it but add only a nondefining declaration for it, usefully invoking the default contract-violation handler would still not be possible because doing so requires a `const` reference to an object of type `std::contracts::contract_violation` to be passed in as a function parameter, but no such object can be constructed by the user because it does not have a public constructor. While such a nondefining declaration would, in theory, allow the user to take the address of the default contract-violation handler, no assumption should ever be made that its address would be the same as the address of `invoke_default_contract_violation_handler`.

SG21, Teleconference, 2024-04-04, Poll 1

For the Contracts MVP, rename `invoke_default_contract_violation_handler` to `default_contract_violation_handler`.

SF	F	N	A	SA
0	2	3	7	4

Result: Consensus against

SG21, Teleconference, 2024-04-04, Poll 2

For the Contracts MVP, rename `invoke_default_contract_violation_handler` to `default_handle_contract_violation`.

SF	F	N	A	SA
0	0	4	9	4

Result: Consensus against

SG21, Teleconference, 2024-04-04, Poll 5

For the Contracts MVP, rename `invoke_default_contract_violation_handler` to `invoke_default_violation_handler`.

SF	F	N	A	SA
3	4	2	6	3

Result: No consensus

3.7.5 Standard Library Contracts

The Library Working Group has been normalizing all Standard Library specifications to use distinct preconditions and postconditions clauses for Standard Library functions. These clauses are often prose but also frequently contain statements of the form "*expression* is `true`." Each of the latter form preconditions and postconditions could be easily transformed into precondition and postcondition assertions.

Making that change, however, would dictate a choice that is currently a quality-of-implementation decision and, more importantly, would prescribe exactly how those contract assertions would be checked by all Standard Library implementations. In some cases, a push has been made to do so, such as the library hardening proposed by [P3471R4]. The problem, however, is that we do not yet have the tools to distinguish contract assertions that should be treated differently in code, and this feature will be needed as more libraries interact. Library hardening handles this by being both optional and not specifying the exact contract assertions that are used to enforce hardened preconditions, leaving the choice of `pre`, `post`, `contract_assert`, or some compiler-provided built-in with added functionality entirely up to the library implementers.

In addition, many libraries already check these conditions using legacy macro-based facilities, and taking those libraries' existing, useful, and effective tools and rendering them immediately nonconforming with a new Standard would be a grave disservice.

Following the concerns raised in [P3173R0], EWG has taken polls on whether Contracts should be applied to the Standard Library. While these polls revealed a strong appetite to gain *usage* experience of contract assertions in a Standard Library implementation (which was subsequently realized; see [P3460R0]), significantly less interest was shown in adding contract assertions to the *specification* of the Standard Library in the C++ Standard (something that we consider unwise, for the reasons stated above).

EWG, Tokyo, 2024-03-20, Poll 9

P2900r6: contracts - there should be some usage experience of contracts in an implementation of the STL (without necessarily having a paper to adopt these changes) before contracts can move to plenary.

SF	F	N	A	SA
16	30	12	3	0

EWG, Tokyo, 2024-03-20, Poll 10

P2900r6: contracts - there should be some usage specification of contracts in the STL before contracts can move to plenary.

SF	F	N	A	SA
6	10	11	14	15

3.8 Feature Test Macros

The feature test macro `__cpp_contracts` for the proposed language feature was added to the Contracts paper in revision [P2388R2]. Initially, this feature test macro was intended to indicate availability of the entire proposed functionality. However, LEWG design review of [P2900R11] in November 2024 in Wrocław revealed that the library API should get a separate feature test macro because a conforming implementation may choose to omit the ability to install a user-defined contract-violation handler, in which case no reason exists to provide the header `<contracts>` and the library API contained therein. That macro, `__cpp_lib_contracts`, was added in revision [P2900R12] as requested by LEWG.

4 Errata

4.1 A Flaw in Section 3.5.3, "Observable Checkpoints"

The code example and the accompanying description given in Section 3.5.3, "Observable Checkpoints," in [P2900R14] contain a flaw. The increment of `i` is not part of the observable prefix when `*p` is evaluated, but it can be part of the undefined behavior that occurs when the null-dereference happens: An access to `i` may occur as part of the "arbitrary additional observable behavior" allowed

following the undefined operation. In other words, for the example program listed in [P2900R14], Section 3.5.3,

```
volatile int i = 0;
void f(int *p) {
  if (p != nullptr)
    ++i;

  contract_assert( *p >= 0 ); // undefined behavior if p == nullptr
}
```

it is conforming to exhibit the behavior of the following program (if the contract assertion is known to be evaluated with a checking semantic):

```
volatile int i = 0;
void f(int *p) {
  ++i;
  contract_assert( *p >= 0 ); // undefined behavior if p == nullptr
}
```

Instead, the following would be a more correct code example illustrating the intended semantics:

```
volatile int i = 0;
void f(int *p) {
  if (p != nullptr)
    ++i;
  else
    --i;

  contract_assert( *p >= 0 ); // undefined behavior if p == nullptr
}
```

In this example, the decrement would be observable prior to the UB because of the observable checkpoint (although the UB could increment it twice and lead to a nearly identical end result as if an unconditional increment had occurred).

Acknowledgments

Thanks to everyone who participated in the many discussions at SG21 meetings and teleconferences and on the SG21 reflector and who thus helped shape this paper and to all who contributed to efforts to introduce a contract-checking feature into C++ prior to the inception of SG21.

Thanks to Hubert Tong for providing the information in the Errata section.

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[Abrahams2023] David Abrahams, "Values: Safety, Regularity, Independence, and the Future of Programming (CppCon 2023)". https://www.youtube.com/watch?v=QthAU-t3PQ4, 2023-01-07

[NSA2022] National Security Agency, "Cybersecurity Information Sheet – Software Memory Safety". https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF, 2022-11

[Bastien2023] JF Bastien, "Safety and Security: The Future of C++ (C++Now 2023 Keynote)". https://www.youtube.com/watch?v=Gh79wcGJdTg, 2023-07-14

[Carruth2023] Chandler Carruth, "Carbon Language Successor Strategy: From C++ Interop to Memory Safety (C++Now 2023 Keynote)". https://www.youtube.com/watch?v=1ZTJ9omXOQ0, 2023-09-08)

[Claburn2023] Thomas Claburn, "Microsoft is busy rewriting core Windows code in memory-safe Rust". https://www.theregister.com/2023/04/27/microsoft_windows_rust/, 2023-04-27

[CISA2023] Cybersecurity and Infrastructure Security Agency, "The Case for Memory Safe Roadmaps". https://www.cisa.gov/sites/default/files/2023-12/T, 2023-12

[Lakshmanan2024] Ravie Lakshmanan, "Google's Shift to Rust Programming Cuts Android Memory Vulnerabilities by 68%". https://thehackernews.com/2024/09/googles-shift-to-rust-programming-cuts.html, 2024-09-25

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

[N1669] Thorsten Ottosen, "Proposal to add Contract Programming to C++ (revision 1)", 2004
http://wg21.link/N1669

[N1773] D. Abrahams, L. Crowl, T. Ottosen, and J. Widman, "Proposal to add Contract Programming to C++ (revision 2)", 2005
http://wg21.link/N1773

[N1866] L. Crowl and T. Ottosen, "Proposal to add Contract Programming to C++ (revision 3)", 2005
http://wg21.link/N1866

[N1962] L. Crowl and T. Ottosen, "Proposal to add Contract Programming to C++ (revision 4)", 2006
http://wg21.link/N1962

[N3604] J. Lakos and A. Zakharov, "Centralized Defensive-Programming Support for Narrow Contracts", 2013
http://wg21.link/N3604

[N3753] J. Lakos and A. Zakharov, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 1)", 2013
http://wg21.link/N3753

[N3818] J. Lakos and A. Zakharov, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 2)", 2013
http://wg21.link/N3818

[N3877] J. Lakos and A. Zakharov, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 3)", 2014
http://wg21.link/N3877

[N3963] J. Lakos and A. Zakharov, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 4)", 2014
http://wg21.link/N3963

[N3997] J. Lakos, A. Zakharov, and A. Beels, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 5)", 2014
http://wg21.link/N3997

[N4075] J. Lakos, A. Zakharov, and A. Beels, "Centralized Defensive-Programming Support for Narrow Contracts (Revision 5)", 2014
http://wg21.link/N4075

[N4110] J. Daniel Garcia, "Exploring the design space of contract specifications for C++", 2014
http://wg21.link/N4110

[N4135] J. Lakos, A. Zakharov, A. Beels, and N. Myers, "Language Support for Runtime Contract Validation (Revision 8)", 2014
http://wg21.link/N4135

[N4248] Alisdair Meredith, "Library Preconditions are a Language Feature", 2014
http://wg21.link/N4248

[N4253] J. Lakos, A. Zakharov, A. Beels, and N. Myers, "Language Support for Runtime Contract Validation (Revision 9)", 2014
http://wg21.link/N4253

[N4293] J. Daniel Garcia, "C++ language support for contract programming", 2014
http://wg21.link/N4293

[N4319] Gabriel Dos Reis, Shuvendu Lahiri, Francesco Logozzo, Thomas Ball, and Jared Parsons, "Contracts for C++: What are the Choices", 2014
http://wg21.link/N4319

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

[N4415] Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, Manuel Fahndrich, and Shuvendu Lahri, "Simple Contracts for C++", 2015
http://wg21.link/N4415

[N4435] Walter Brown, "Proposing Contract Attributes", 2015
http://wg21.link/N4435

[P0147R0] Lawrence Crowl, "The Use and Implementation of Contracts", 2015
http://wg21.link/P0147R0

[P0166R0] J. Daniel Garcia, "Three interesting questions about contracts", 2015
http://wg21.link/P0166R0

[P0246R0] Nathan Myers, "Contract Assert Support Merged Proposal", 2016
http://wg21.link/P0246R0

[P0247R0] Nathan Myers, "Criteria for Contract Support", 2016
http://wg21.link/P0247R0

[P0265R0] Michael Wong, "SG5 is NOT proposing Transactional Memory for C++17", 2016
http://wg21.link/P0265R0

[P0287R0] Gabriel Dos Reis, "Simple Contracts for C++", 2016
http://wg21.link/P0287R0

[P0380R0] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "A Contract Design", 2016
http://wg21.link/P0380R0

[P0380R1] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "A Contract Design", 2016
http://wg21.link/P0380R1

[P0465R0] Lisa Lippincott, "Procedural Function Interfaces", 2016
http://wg21.link/P0465R0

[P0542R0] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "Support for contract based programming in C++", 2017
http://wg21.link/P0542R0

[P0542R1] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "Support for contract based programming in C++", 2017
http://wg21.link/P0542R1

[P0542R2] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "Support for contract based programming in C++", 2017
http://wg21.link/P0542R2

[P0542R4] J. Daniel Garcia, "Support for contract based programming in C++", 2018
http://wg21.link/P0542R4

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

[P1208R6] Corentin Jabot, Robert Douglas, Daniel Krügler, and Peter Sommerlad, "Adopt source location from Library Fundamentals V3 for C++20", 2019
http://wg21.link/P1208R6

[P1289R1] J. Daniel Garcia and Ville Voutilainen, “Access control in contract conditions”, 2018
http://wg21.link/P1289R1

[P1290R3] J. Daniel Garcia and Ville Voutilainen, “Avoiding undefined behavior in contracts”, 2019
http://wg21.link/P1290R3

[P1320R2] Ville Voutilainen, “Allowing contract predicates on non-first declarations”, 2019
http://wg21.link/P1320R2

[P1321R0] Ville Voutilainen, “UB in contract violations”, 2018
http://wg21.link/P1321R0

[P1323R0] Hubert S.K. Tong, “Contract postconditions and return type deduction”, 2018
http://wg21.link/P1323R0

[P1323R2] Hubert S.K. Tong, “Contract postconditions and return type deduction”, 2019
http://wg21.link/P1323R2

[P1332R0] Joshua Berne, Nathan Burgers, Hyman Rosen, and John Lakos, “Contract Checking in C++: A (long-term) Road Map”, 2018
http://wg21.link/P1332R0

[P1333R0] Joshua Berne and John Lakos, “Assigning Concrete Semantics to Contract-Checking Levels at Compile Time”, 2018
http://wg21.link/P1333R0

[P1334R0] Joshua Berne and John Lakos, “Specifying Concrete Semantics Directly in Contract-Checking Statements”, 2018
http://wg21.link/P1334R0

[P1344R1] Nathan Myers, “Pre/Post vs. Enspects/Exsures”, 2019
http://wg21.link/P1344R1

[P1429R3] Joshua Berne and John Lakos, “Contracts That Work”, 2019
http://wg21.link/P1429R3

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

[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall, “Minimizing Contracts”, 2019
http://wg21.link/P1607R1

[P1639R0] Corentin Jabot, “Unifying source_location and contract_violation”, 2019
http://wg21.link/P1639R0

[P1670R0] Joshua Berne and Alisdair Meredith, “Side Effects of Checked Contracts and Predicate Elision”, 2019
http://wg21.link/P1670R0

[P1671R0] Joshua Berne and Alisdair Meredith, “Contract Evaluation in Constant Expressions”, 2019
http://wg21.link/P1671R0

[P1680R0] Andrew Sutton and Jeff Chapman, “Implementing Contracts in GCC”, 2019
http://wg21.link/P1680R0

[P1710R0] Ville Voutilainen, “Adding a global contract assumption mode”, 2019
http://wg21.link/P1710R0

[P1711R0] Bjarne Stroustrup, “What to do about contracts?”, 2019
http://wg21.link/P1711R0

[P1730R0] Hyman Rosen, John Lakos, and Alisdair Meredith, “Adding a global contract assumption mode”, 2019
http://wg21.link/P1730R0

[P1743R0] Rostislav Khlebnikov and John Lakos, “Contracts, Undefined Behavior, and Defensive Programming”, 2019
http://wg21.link/P1743R0

[P1744R0] Rostislav Khlebnikov and John Lakos, “Avoiding Misuse of Contract-Checking”, 2019
http://wg21.link/P1744R0

[P1769R0] Ville Voutilainen, “The "default" contract build-level and continuation-mode should be implementation-defined”, 2019
http://wg21.link/P1769R0

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

[P1786R0] Hyman Rosen, John Lakos, and Alisdair Meredith, “Adding a global contract assumption mode”, 2019
http://wg21.link/P1786R0

[P1812R0] Timur Doumler and Ville Voutilainen, “Axioms should be assumable: a minimal fix for contracts”, 2019
http://wg21.link/P1812R0

[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

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

[P2012R2] Nicolai Josuttis, Victor Zverovich, Arthur O’Dwyer, and Filipe Mulonde, “Fix the range-based for loop, Rev2”, 2021
http://wg21.link/P2012R2

[P2026R0] Ryan McDougall, Bryce Adelstein Lelbach, JF Bastien, Andreas Weis, Ruslan Arutyunyan, and Ilya Burylov, "A Constituent Study Group for Safety-Critical Applications", 2020
http://wg21.link/P2026R0

[P2032R0] Joshua Berne, "Contracts - What Came Before", 2020
http://wg21.link/P2032R0

[P2036R1] Barry Revzin, "Changing scope for lambda trailing-return-type", 2021
http://wg21.link/P2036R1

[P2036R3] Barry Revzin, "Changing scope for lambda trailing-return-type", 2021
http://wg21.link/P2036R3

[P2038R0] Andrzej Krzemieński and Ryan McDougall, "Proposed nomenclature for contract-related proposals", 2020
http://wg21.link/P2038R0

[P2053R0] Rostislav Khlebnikov and John Lakos, "Defensive Checks Versus Input Validation", 2020
http://wg21.link/P2053R0

[P2053R1] Rostislav Khlebnikov and John Lakos, "Defensive Checks Versus Input Validation", 2020
http://wg21.link/P2053R1

[P2064R0] Herb Sutter, "Assumptions", 2020
http://wg21.link/P2064R0

[P2076R0] Ville Voutilainen, "Previous disagreements on Contracts", 2020
http://wg21.link/P2076R0

[P2114R0] Joshua Berne, Ryan McDougall, and Andrzej Krzemieński, "Minimial Contract Use Cases", 2020
http://wg21.link/P2114R0

[P2173R1] Daveed Vandevoorde, Inbal Levi, and Ville Voutilainen, "Attributes on Lambda-Expressions", 2021
http://wg21.link/P2173R1

[P2182R1] Andrzej Krzemieński, Joshua Berne, and Ryan McDougall, "Contract Support: Defining the Minimum Viable Feature Set", 2020
http://wg21.link/P2182R1

[P2185R0] Caleb Sunstrum, "Contracts Use Case Categorization", 2020
http://wg21.link/P2185R0

[P2339R0] Andrzej Krzemieński, "Contract violation handlers", 2021
http://wg21.link/P2339R0

[P2388R0] Andrzej Krzemieński and Gašper Ažman, "Abort-only contract support", 2021
http://wg21.link/P2388R0

[P2388R1] Andrzej Krzemieński and Gašper Ažman, “Minimum Contract Support: either Ignore or Check_and_abort”, 2021
http://wg21.link/P2388R1

[P2388R2] Andrzej Krzemieński and Gašper Ažman, “Minimum Contract Support: either Ignore or Check_and_abort”, 2021
http://wg21.link/P2388R2

[P2388R3] Andrzej Krzemieński and Gašper Ažman, “Minimum Contract Support: either No_eval or Eval_and_abort”, 2021
http://wg21.link/P2388R3

[P2388R4] Andrzej Krzemieński and Gašper Ažman, “Minimum Contract Support: either No_eval or Eval_and_abort”, 2021
http://wg21.link/P2388R4

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki, “Closure-based Syntax for Contracts”, 2021
http://wg21.link/P2461R1

[P2466R0] Andrzej Krzemieński, “The notes on contract annotations”, 2021
http://wg21.link/P2466R0

[P2487R1] Andrzej Krzemieński, “Is attribute-like syntax adequate for contract annotations?”, 2023
http://wg21.link/P2487R1

[P2510R2] Mark de Wever, “Formatting pointers”, 2022
http://wg21.link/P2510R2

[P2521R4] Andrzej Krzemieński, “Contract support – Record of SG21 consensus”, 2023
http://wg21.link/P2521R4

[P2521R5] Andrzej Krzemieński, “Contract support – Record of SG21 consensus”, 2023
http://wg21.link/P2521R5

[P2570R2] Andrzej Krzemieński, “Contract predicates that are not predicates”, 2023
http://wg21.link/P2570R2

[P2573R0] Yihe Li, “= delete("should have a reason");”, 2022
http://wg21.link/P2573R0

[P2646R0] Parsa Amini, Joshua Berne, and John Lakos, “Explicit Assumption Syntax Can Reduce Run Time”, 2022
http://wg21.link/P2646R0

[P2659R2] Brian Bi and Alisdair Meredith, “A Proposal to Publish a Technical Specification for Contracts”, 2023
http://wg21.link/P2659R2

[P2680R0] Gabriel Dos Reis, “Contracts for C++: Prioritizing Safety”, 2022
http://wg21.link/P2680R0

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

[P2688R3] Michael Park, “Pattern Matching: `match` Expression”, 2024
http://wg21.link/P2688R3

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

[P2698R0] Bjarne Stroustrup, “Unconditional termination is a serious problem”, 2022
http://wg21.link/P2698R0

[P2700R0] Timur Doumler, Andrzej Krzemieński, John Lakos, Joshua Berne, Brian Bi, Peter Brett, Oliver Rosten, and Herb Sutter, “Questions on P2680 "Contracts for C++: Prioritizing Safety"”, 2022
http://wg21.link/P2700R0

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

[P2737R0] Andrew Tomazos, “Proposal of Condition-centric Contracts Syntax”, 2023
http://wg21.link/P2737R0

[P2743R0] Gabriel Dos Reis, “Contracts for C++: Prioritizing Safety - Presentation slides of [P2680R0]”, 2023
http://wg21.link/P2743R0

[P2750R2] Jarrad J. Waterloo, “C Dangling Reduction”, 2023
http://wg21.link/P2750R2

[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

[P2756R0] Andrew Tomazos, “Proposal of Simple Contract Side Effect Semantics”, 2023
http://wg21.link/P2756R0

[P2771R1] Thomas Neumann, “Towards memory safety in C++”, 2023
http://wg21.link/P2771R1

[P2784R0] Andrzej Krzemieński, “Not halting the program after detected contract violation”, 2023
http://wg21.link/P2784R0

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

[P2809R3] JF Bastien, “Trivial infinite loops are not Undefined Behavior”, 2024
http://wg21.link/P2809R3

[P2811R5] Joshua Berne, “Contract-Violation Handlers”, 2023
http://wg21.link/P2811R5

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

[P2828R0] Brian Bi, “Copy elision for direct-initialization with a conversion function (Core issue 2327)”, 2023
http://wg21.link/P2828R0

[P2829R0] Andrew Tomazos, “Proposal of Contracts Supporting Const-On-Definition Style”, 2023
http://wg21.link/P2829R0

[P2831R0] Timur Doumler and Ed Catmur, “Functions having a narrow contract should not be noexcept”, 2023
http://wg21.link/P2831R0

[P2834R0] Joshua Berne and John Lakos, “Semantic Stability Across Contract-Checking Build Modes”, 2023
http://wg21.link/P2834R0

[P2834R1] Joshua Berne and John Lakos, “Semantic Stability Across Contract-Checking Build Modes”, 2023
http://wg21.link/P2834R1

[P2838R0] Ville Voutilainen, “Unconditional contract violation handling of any kind is a serious problem”, 2023
http://wg21.link/P2838R0

[P2852R0] Tom Honermann, “Contract violation handling semantics for the contracts MVP”, 2023
http://wg21.link/P2852R0

[P2853R0] Andrew Tomazos, “Proposal of std::contract_violation”, 2023
http://wg21.link/P2853R0

[P2858R0] Andrzej Krzemieński, “Noexcept vs contract violations”, 2023
http://wg21.link/P2858R0

[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

[P2890R1] Timur Doumler, “Contracts on lambdas”, 2023
http://wg21.link/P2890R1

[P2890R2] Timur Doumler, “Contracts on lambdas”, 2023
http://wg21.link/P2890R2

[P2894R2] Timur Doumler, “Constant evaluation of Contracts”, 2024
http://wg21.link/P2894R2

[P2900R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2023
http://wg21.link/P2900R0

[P2900R10] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R10

[P2900R11] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R11

[P2900R12] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R12

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

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

[P2900R4] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R4

[P2900R5] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R5

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

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R7

[P2900R8] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R8

[P2900R9] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R9

[P2932R2] Joshua Berne, “A Principled Approach to Open Design Questions for Contracts”, 2023
http://wg21.link/P2932R2

[P2932R3] Joshua Berne, “A Principled Approach to Open Design Questions for Contracts”, 2024
http://wg21.link/P2932R3

[P2935R4] Joshua Berne, “An Attribute-Like Syntax for Contracts”, 2023
http://wg21.link/P2935R4

[P2954R0] Ville Voutilainen, “Contracts and virtual functions for the Contracts MVP”, 2023
http://wg21.link/P2954R0

[P2957R0] Andrzej Krzemieński and Iain Sandoe, “Contracts and coroutines”, 2023
http://wg21.link/P2957R0

[P2957R1] Andrzej Krzemieński and Iain Sandoe, “Contracts and coroutines”, 2024
http://wg21.link/P2957R1

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

[P2961R2] Timur Doumler and Jens Maurer, “A natural syntax for Contracts”, 2023
http://wg21.link/P2961R2

[P2969R0] Timur Doumler, Ville Voutilainen, and Tom Honermann, “Contract annotations are potentially-throwing”, 2023
http://wg21.link/P2969R0

[P2971R2] Walter E Brown, “Implication for C++”, 2024
http://wg21.link/P2971R2

[P3007R0] Timur Doumler, Andrzej Krzemieński, and Joshua Berne, “Return object semantics in postconditions”, 2023
http://wg21.link/P3007R0

[P3028R0] Joshua Berne, Gašper Ažman, Rostislav Khlebnikov, and Timur Doumler, “An Overview of Syntax Choices for Contracts”, 2023
http://wg21.link/P3028R0

[P3066R0] Timur Doumler, “Allow repeating contract annotations on non-first declarations”, 2023
http://wg21.link/P3066R0

[P3071R0] Jens Maurer, “Protection against modifications in contracts”, 2023
http://wg21.link/P3071R0

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

[P3073R0] Timur Doumler and Ville Voutilainen, “Remove evaluation_undefined_behavior and will_continue from the Contracts MVP”, 2024
http://wg21.link/P3073R0

[P3079R0] Oliver Rosten, “Should ignore and observe exist for constant evaluation of contracts?”, 2024
http://wg21.link/P3079R0

[P3081R0] Herb Sutter, “Core safety Profiles: Specification, adoptability, and impact”, 2024
http://wg21.link/P3081R0

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

[P3081R2] Herb Sutter, “Core safety profiles for C++26”, 2025
http://wg21.link/P3081R2

[P3088R1] Timur Doumler and Joshua Berne, “Attributes for contract assertions”, 2024
http://wg21.link/P3088R1

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

[P3098R0] Timur Doumler, Gašper Ažman, and Joshua Berne, “Contracts for C++: Post-condition captures”, 2024
http://wg21.link/P3098R0

[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

[P3102R0] Joshua Berne, “Refining Contract Violation Detection Modes”, 2024
http://wg21.link/P3102R0

[P3113R0] Timur Doumler, “Slides: Contract assertions, the noexcept operator, and deduced exception specifications”, 2024
http://wg21.link/P3113R0

[P3114R0] Andrzej Krzemieński, “noexcept(contract_assert(_)) – slides”, 2024
http://wg21.link/P3114R0

[P3119R0] Joshua Berne, “Tokyo Technical Fixes to Contracts”, 2024
http://wg21.link/P3119R0

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

[P3165R0] Ville Voutilainen, “Contracts on virtual functions for the Contracts MVP”, 2024
http://wg21.link/P3165R0

[P3167R0] Tom Honermann, “Attributes for the result name in a postcondition assertion”, 2024
http://wg21.link/P3167R0

[P3169R0] Jonas Persson, “Inherited contracts”, 2024
http://wg21.link/P3169R0

[P3172R0] Andrzej Krzemieński, “Using `this` in constructor preconditions”, 2024
http://wg21.link/P3172R0

[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

[P3197R0] Timur Doumler and John Spicer, “A response to the Tokyo EWG polls on the Contracts MVP ([P2900R6])”, 2024
http://wg21.link/P3197R0

[P3198R0] Andrzej Krzemieński, “A takeaway from the Tokyo LEWG meeting on Contracts MVP”, 2024
http://wg21.link/P3198R0

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

[P3210R0] Andrew Tomazos, “A Postcondition *is* a Pattern Match”, 2024
http://wg21.link/P3210R0

[P3210R2] Andrew Tomazos, “A Postcondition *is* a Pattern Match”, 2024
http://wg21.link/P3210R2

[P3221R0] Jonas Persson, “Disable pointers to contracted functions”, 2024
http://wg21.link/P3221R0

[P3226R0] Timur Doumler, “Contracts for C++: Naming the "Louis semantic"”, 2024
http://wg21.link/P3226R0

[P3227R1] Gašper Ažman and Timur Doumler, “Fixing the library API for contract violation handling”, 2024
http://wg21.link/P3227R1

[P3228R1] Timur Doumler, “Contracts for C++: Revisiting contract check elision and duplication”, 2024
http://wg21.link/P3228R1

[P3229R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Making erroneous behaviour compatible with Contracts”, 2025
http://wg21.link/P3229R0

[P3237R2] Andrei Zissu, “Matrix Representation of Contract Semantics”, 2024
http://wg21.link/P3237R2

[P3238R0] Ville Voutilainen, “An alternate proposal for naming contract semantics”, 2024
http://wg21.link/P3238R0

[P3249R0] Ran Regev, “A unified syntax for Pattern Matching and Contracts when introducing a new name”, 2024
http://wg21.link/P3249R0

[P3250R0] Peter Bindels, “C++ contracts with regards to function pointers”, 2024
http://wg21.link/P3250R0

[P3251R0] Peter Bindels, “C++ contracts and coroutines”, 2024
http://wg21.link/P3251R0

[P3257R0] Jens Maurer, “Make the predicate of contract_assert more regular”, 2024
http://wg21.link/P3257R0

[P3261R1] Joshua Berne, “Revisiting `const`-ification in Contract Assertions”, 2024
http://wg21.link/P3261R1

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

[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

[P3270R0] John Lakos and Joshua Berne, “Repetition, Elision, and Constification w.r.t. contract_assert”, 2024
http://wg21.link/P3270R0

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

[P3274R0] Bjarne Stroustrup, “A framework for Profiles development”, 2024
http://wg21.link/P3274R0

[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

[P3281R0] John Spicer, "Contact checks should be regular C++", 2024
http://wg21.link/P3281R0

[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

[P3297R1] Ryan McDougall, Jean-Francois Campeau, Christian Eltzschig, Mathias Kraus, and Pez Zarifian, "C++26 Needs Contract Checking", 2024
http://wg21.link/P3297R1

[P3316R0] Jonas Persson, "A more predictable unchecked semantic", 2024
http://wg21.link/P3316R0

[P3318R0] Ville Voutilainen, "Throwing violation handlers, from an application programming perspective", 2024
http://wg21.link/P3318R0

[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

[P3328R0] Joshua Berne, "Observable Checkpoints During Contract Evaluation", 2024
http://wg21.link/P3328R0

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

[P3338R0] Ville Voutilainen, "Observe and ignore semantics in constant evaluation", 2024
http://wg21.link/P3338R0

[P3344R0] Joshua Berne, Timur Doumler, and Lisa Lippincott, "Virtual Functions on Contracts (EWG - Presentation for P3097)", 2024
http://wg21.link/P3344R0

[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

[P3386R0] Joshua Berne, “Static Analysis of Contracts with P2900”, 2024
http://wg21.link/P3386R0

[P3387R0] Timur Doumler, Joshua Berne, Iain Sandoe, and Peter Bindels, “Contract assertions on coroutines”, 2024
http://wg21.link/P3387R0

[P3390R0] Sean Baxter and Christian Mazakas, “Safe C++”, 2024
http://wg21.link/P3390R0

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

[P3417R1] Gašper Ažman and Timur Doumler, “Improving the handling of exceptions thrown from contract predicates”, 2025
http://wg21.link/P3417R1

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

[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

[P3483R0] Timur Doumler and Joshua Berne, “Contracts for C++: Pre-Wroclaw technical clarifications”, 2024
http://wg21.link/P3483R0

[P3483R1] Timur Doumler and Joshua Berne, “Contracts for C++: Pre-Wroclaw technical clarifications”, 2024
http://wg21.link/P3483R1

[P3484R1] Timur Doumler and Joshua Berne, “Postconditions odr-using a parameter modified in an overriding function”, 2024
http://wg21.link/P3484R1

[P3484R2] Timur Doumler and Joshua Berne, “Postconditions odr-using a parameter modified in an overriding function”, 2024
http://wg21.link/P3484R2

[P3487R0] Timur Doumler and Joshua Berne, “Postconditions odr-using a parameter that may be passed in registers”, 2024
http://wg21.link/P3487R0

[P3489R0] Timur Doumler and Joshua Berne, “Postconditions odr-using a parameter of dependent type”, 2024
http://wg21.link/P3489R0

[P3499R0] Lisa Lippincott, Timur Doumler, and Joshua Berne, “Exploring strict contract predicates”, 2025
http://wg21.link/P3499R0

[P3500R0] Timur Doumler, Gašper Ažman, and Joshua Berne, “Are Contracts "safe"?”, 2025
http://wg21.link/P3500R0

[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

[P3510R2] Nathan Myers and Gašper Ažman, “Leftover properties of `this` in constructor preconditions”, 2024
http://wg21.link/P3510R2

[P3520R0] Timur Doumler, Joshua Berne, and Andrzej Krzemieński, “Wroclaw Technical Fixes to Contracts”, 2024
http://wg21.link/P3520R0

[P3541R1] Andrzej Krzemieński, “Violation handlers vs `noexcept`”, 2025
http://wg21.link/P3541R1

[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

[P3577R0] John Lakos, “Require a non-throwing default contract-violation handler”, 2025
http://wg21.link/P3577R0

[P3578R0] Ryan McDougall, “Language Safety and Grandma Safety”, 2025
http://wg21.link/P3578R0

[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

[P3599R0] Joshua Berne and Timur Doumler, “Contextualizing Contracts Concerns”, 2025
http://wg21.link/P3599R0

[P3626R0] Timur Doumler, “Make predicate exceptions propagate by default”, 2025
http://wg21.link/P3626R0

[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

[CR2023] Consumer Reports, “Report: Future of Memory Safety”. https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report-1-1.pdf, 2023-01-22