Contextualizing Contracts Concerns

Document #:	P3591R0
Date:	2025-02-03
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <papers@timur.audio>

Abstract

Recent papers, [P3573R0] and [P3506R0], have listed several concerns related to the Contracts MVP, [P2900R13]. Readers of those papers might mistakenly conclude that these concerns are new and profound flaws in the proposed Contracts facility and have not been previously discussed and addressed. In this paper, we present the missing background and facts related to each of the concerns raised; we describe how all the concerns have been discussed, how consensus was reached on their solutions, and why [P2900R13] is more than ready to be included in the C++ Standard.

Contents

Revision History

Revision 0

- Original version of the paper in response to [P3573R0] and [P3506R0]

1 Introduction

As chronicled in [P2899R0], the Contracts proposal in [P2900R13] is the culmination of a huge collective investment of time and consensus building by many participants in WG21.

The authors of [P3573R0] and [P3506R0], while undoubtedly being sincere and passionate stewards of the language, have resurrected concerns that have already been repeatedly discussed. Those discussions led to decisions, made with strong consensus, that are incorporated into the Contracts MVP, [P2900R13]. Our standard procedure within WG21 is, of course, to respond to all concerns with thought and care, yet that process must begin by first reviewing the history of the discussion of that concern, deciding if the earlier discussion was sufficient, and determining if any new information about a concern warrants re-evaluating the earlier decisions of the Committee.

In this paper, we will present a starting point for the history of the detailed discussions that led to the design in [P2900R13] and explain — largely for those who have not been involved in discussions and might believe that the concerns in [P3573R0] and [P3506R0] have not been considered — that none of the raised issues are reasons to be opposed to adopting [P2900R13] for C++26. Along the way, readers will clearly see that the presented concerns are self-contradictory and that the strongly held consensus for [P2900R13] is the result of diligent effort to find the right balance between the technical solutions available for the very concerns raised in [P3573R0] and [P3506R0].

2 Concerns

We first address the concerns that were presented in [P3573R0]. We list [P3573R0]'s concerns in the same order it did (which it describes as random), and we provide information, for each concern in that paper, to help readers understand the context for the concern and its related discussion and decisions in the Committee.

- **`const`-ification** — We wish to discourage, within the predicate of a contract assertion, modifications to variables from outside the contract assertion. To do so, expressions that denote such variables are made `const`, a feature that has come to be called `const`-ification. Since its introduction into the Contracts MVP, this feature of [P2900R13] has been discussed thoroughly in both SG21 and EWG. As described in Section 3.4.2 of [P2899R0], the history of this aspect of the design is fairly long and goes back many years.

  [P3071R1] introduced `const`-ification into the Contracts MVP in December 2023. This initial proposal, when reviewed in SG21, was met with overwhelming support.

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

For the Contracts MVP, adopt [P3071R1] "Protection against modifications in contracts" as presented.

SF	F	N	A	SA
6	10	3	0	0

Result: Consensus

When [P2900R7] was forwarded to EWG, the question came up again as to whether `const`-ification should be retained. The wider group did not have strong consensus to either keep or remove `const`-ification, so efforts to analyze the impact of the feature were undertaken. In particular, [P3261R2] was produced to thoroughly explore alternative approaches. With each discussion of [P3261R2], twice in SG21 and then EWG, consensus to keep `const`-ification increased; learning about the feature and its ramifications clarifies its real upsides and lack of meaningful downsides. The most recent discussion in Wrocław resulted in clear consensus to keep `const`-ification in the Contracts MVP.

EWG, Wrocław, 2024-11-19, Poll

[P3261R1] / [P3478R0]: [P2900R8] shall not have `const`-ification by default.

SF	F	N	A	SA
10	4	9	19	12

Result: Consensus against

The repeated and extensive discussion of `const`-ification has shown that opposition to it has reduced over time; the right solution in the eyes of the majority of those involved in standardizing the core language and Contracts is to keep `const`-ification.

[P3573R0] attempts to resurrect a few more specific issues related to `const`-ification that the discussions have already addressed.

- Both implementations of Contracts currently in progress have expressed that the implementation of `const`-ification, including quality error messages, was a minuscule effort.
- As described in [P3261R2], the feature, when applied to large codebases using contract assertions, catches errors and finds real bugs and flaws in existing production code. The flowchart described in Section 2.3.1 clarifies that the situations in which `const`-ification does actual harm are exceedingly rare, and the case studies summarized with pie charts in Section 2.3.2 show how this takes effect in practice.

  More importantly, based on the opinions of those with experience supporting real contract-checking facilities today, `const`-ification would prevent a significant portion of the problems that confound users. Problems that arise when a contract assertion's predicate makes destructive changes to program state can be exceedingly hard to diagnose since not one program but two (the build with assertions checked and the one with them unchecked) must be analyzed to identify and fix the culprit. In addition, cataloging this form of problem is challenging because the problems themselves often extend the development

time significantly but do not escape code review and testing and thus do not appear in a publicly available repository.

– Understanding `const`-ification is shockingly trivial, so teachability is not a concern. A novice need only know that variables declared outside a contract predicate will be treated as `const` within the predicate, in the same manner that member variables are treated as `const` when accessed within a `const` member function.

  More to the point, having this knowledge is unnecessary when first learning to write contract assertions since most programmers following the simple (and vital) guideline of not writing contract predicates that modify state will rarely notice that `const`-ification is happening. When such users do notice, the reason their attempt to modify state is misguided will be obvious, and they will redirect their efforts appropriately.

– As described in [P3261R2] and contrary to the false claim in [P3573R0], an equivalent warning to `const`-ification has been shown to be unimplementable,[1] and those cases that can be implemented as a warning will simply not catch many categories of errors that `const`-ification will catch. In particular, attempts to *correct* a violation within a predicate by correcting the state of data structures are much less likely to be caught by a warning. Consider, for example, a function that expects a sorted vector as input:

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

  Here, `const`-ification is being applied to `v`, resulting in the `const` overloads of `begin` and `end` being invoked. Notably, the author of `f` simply does not care since the `const` and non-`const` overloads of these member functions of `std::vector` have the same behavior; the only difference between the overloads is whether their return values enable modification of the contents of the container.

  A programmer who does care comes along later and decides that this defensive check should try to *fix* the problem by sorting the vector when an unsorted vector is encountered:

```
void f(std::vector<int> v)
  pre(( std::sort(v.begin(), v.end()), true ));  // Predicate is always true.
```

  Such a program will work great with no violations detected in a checked build and then proceed to fail horribly in a released build that turns off (ignores) expensive contract assertions. From experience, this form of bug that disappears when contract checking is turned on is among the most difficult to diagnose and results in significant reduction in the efficacy of the Contracts feature as a whole. None of the alternative proposals for `const`-ification explored in Section 3.1 of [P3261R2] will accept the valid precondition that the vector is sorted and reject the precondition that attempts to sort the vector.

  Discouraging such misguided uses of Contracts is thus a primary benefit of `const`-ification.

[1] Doing speculative analysis of an expression to determine if it would be valid with `const`-ification can easily double the cost of compiling such expressions. In addition, if restricted to being a warning, any side effects of such an analysis, such as template instantiation, would need to be unwound lest they impact the semantics of the rest of the program, which a warning never has leeway to do.

– [P3573R0] claims that `const`-ification does a poor job of identifying modifications, in particular when making modifications that involve dereferencing a provided pointer. Having any form of `const`-ification apply to the result of dereferencing operations — and, in particular, to apply correctly to the results with user-defined types as well as built-in ones — would inevitably introduce a new flavor of `const`-ness that applies *deeply* in a user-defined manner. Without a new, wide-reaching, highly invasive language feature to define *deep* `const`, the distinctions between functions that create new objects that can be modified within a contract assertion and those that return references to external objects simply cannot be made.

  Neither SG21 nor any other group has shown interest in pursuing any form of *deep* `const`, which would be needed to specify such restrictions, although [P3261R2] does explore many aspects of such a potential feature. SG21 discussed deep `const` extensively and agreed with the conclusions of [P3261R2] that deep `const` should not be pursued for Contracts.

- **Exception capture** — Not allowing exceptions to escape from the evaluation of contract predicates, since they always terminated, was a feature of C++2a Contracts. Discussing this aspect of the evaluation of contract assertions revealed that some have a real need for the full spectrum of alternatives when exceptions escape from the evaluation of a predicate.

  – Not allowing exceptions to escape from a contract assertion's predicate is necessary to prevent major misuses of any contract-checking facility, namely to allow client code to learn pragmatically whether contract checking is currently enabled. For example, consider a contract assertion whose sole purpose is to throw an exception whenever it is actively checking, thereby enabling a user to detect that it (and likely surrounding assertions) are currently being actively checked, and take some distinct action accordingly. By not allowing an exception to escape a contract assertion's predicate directly, we prevent the evaluation of contract assertions from influencing local control flow when no contract violation has occurred.

  – Certain classes of exception, such as `bad_alloc`, indicate a general issue with execution, not a problem with a particular contract. Some systems attempt to recover from these exceptions by allowing them to propagate up the stack, and in such systems, outright preventing throwing would be detrimental.

  – The default mode for the language to prevent unwanted exceptions in certain contexts is to terminate when exceptions do escape, a choice of unconditional terminate that violates the concerns raised earlier, in [P2698R0], by one of the authors of [P3573R0].

  To allow both sets of users to see their needs met, the contract-violation handler, as proposed in [P2811R7], is invoked when an exception escapes the predicate. This violation handler is given the full range of options, including terminating (as the C++2a Contracts facility would have done) or rethrowing the exception, which enables the behavior that [P3573R0] requests.

  [P3506R0] raises a concern with the cost of introducing exception handling in these circumstances. Such a cost is potentially non-negligible and must be considered in the context of how contract assertions will actually be used and written.

- The vast majority of contract-assertion predicates are simple expressions, e.g., tests against `nullptr`, numeric ranges, and so on. Each of these — and, in general, any expression that the compiler is going to inline — can easily be identified as to whether it is potentially throwing. If it isn't potentially throwing, then all exception-handling scaffolding will simply go away.
- When a predicate does involve calling into an opaque non-`noexcept` function, making that function call already has significant overhead. The extra cost of handling exceptions that such a call might emit should not be overwhelming; exceptions on most platforms are already highly optimized for the happy path where they are not thrown, and any extra generated code is cold code with almost no overhead.
- We expect compiler vendors to provide mechanisms to explicitly say that a TU (translation unit) is being built such that exceptions will *not* escape the contract-violation handler. In such cases, the majority of the exception-related overhead associated with contract-assertion code generation is expected to disappear.

No other options for handling predicates that throw exceptions will successfully avoid disenfranchising large portions of C++'s user base.

- **Overly complex hierarchy contract model** — The approach to handling contracts on virtual functions is described in detail in [P3097R0]. This design was originally conceived years ago, but pursuing it for the Standard was intentionally delayed until after the Contracts MVP due to many misconceptions about how contract checking and virtual functions should interact. In Tokyo, a compiler vendor pressured this Committee (in [P3173R0]) into growing the Contracts MVP to include virtual functions (among other things), so the design was pursued to completion.

  SG21 discussed this proposal in depth in multiple telecons and at the in-person meeting in St. Louis. After significant discussion on the topic, SG21 reached strong consensus on the design in [P3097R0].

  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

  On the back of this confident approval of the design for virtual functions, a presentation ([P3344R0]) was made to EWG that resulted in strong consensus in that subgroup for this design.

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

Anyone experienced with polls in EWG can recognize these results as being well above the normal bar for strong consensus for a feature. As was clear in the discussion, the core idea of [P3097R0], when presented simply, is easy to understand and reason about. When presented with a virtual function invocation, a novice need only learn to check both sets of function contract assertions — those visible to the caller and those visible to the callee. This approach to the issue provides a way to satisfy all forms of inheritance relationships described in [P3097R0].

[P3506R0] claims that the solution proposed by [P3097R0] must somehow be simplified and ignores the real-world use cases described by [P3097R0], such as multiple inheritance, that are simply broken by any other model.

- **Instability of overall design** — SG21 is filled with passionate, opinionated people, discussing in incredible detail a seemingly simple yet incredible subtle and deep topic. This culture leads to extensive discussions and papers on every aspect of the feature.

  Since adopting the plan for a Contracts MVP, [P2695R1], SG21 has made changes to the Contracts MVP only when a potential update garners strong consensus; SG21 has a paper trail backing those decisions. This process has spawned many papers and discussions, all with the intent of ensuring a clear, documented history behind the reasoning in the proposal we put forth. The lack of such documentation was a major deficit and a cause for problems in C++2a Contracts.

  Since February 2024, when SG21 finalized their desired feature set for the Contracts MVP in [P2900R6] and forwarded it to EWG, fairly few changes have been made to the proposal.

  – In [P2900R7], the *quick-enforce* semantic was added after having been proposed by [P3191R0].

  – In [P2900R8], support for virtual functions as proposed by [P3097R0] was added.

– In [P2900R10], support for contract assertions on coroutines was added.[2]

– In [P2900R11], minor additions were made to the library API, `is_terminating` and `evaluation_exception`.

All the major changes were in response to requests from EWG, and the minor changes were mainly corrections to issues discovered during wording review.

For an ongoing proposal being actively discussed and with a page count as significant as that of [P2900R13], [P3573R0]'s implication that the churn of the Contracts MVP is overwhelming seems unrealistic. The desire expressed in [P3573R0] to see less churn is also completely at odds with the expectation for more features (such as grouping contracts or contracts on function pointers) to be forced into the same proposal; this contradiction is evident in many of [P3573R0]'s concerns.

- **Far too much is implementation defined** — The Contracts MVP includes several aspects that are implementation defined, but all are along boundaries where specific behaviors must be implementation defined by necessity.

  Let us consider the full list of items that are implementation defined in [P2900R13].

  1. The specific mode of termination used by the *enforce* and *quick-enforce* semantics
  2. The exact behavior of the default contract-violation handler
  3. Whether the contract-violation handler is replaceable
  4. The evaluation semantics chosen for each contract assertion evaluation
  5. The maximum number of repeated evaluations of a contract assertion

  The first two items are behaviors of processes that simply do not have a consistent definition that is best for all potential C++ platforms. In particular, program termination can happen in widely different ways, each with distinct tradeoffs; restricting all platforms to a single common option would benefit neither the proposal nor users. Similarly, methods — and specific formatting — for emitting diagnostics from a default contract-violation handler differ for various platforms. The various options are discussed in [P3520R0], along with details of how all the available choices are best for different scenarios at different times. By allowing implementations the freedom to do what is best in their environments, no users are disenfranchised or unduly restricted.

  The replaceability of the contract-violation handler, item 3, is viewed by some platforms as a security risk (where implementors in the past have insisted that they not be required to support such options), yet other users see it as a necessary feature for even the most basic viability of the proposal. To provide a specification viable for both groups, we must give implementations freedom to decline to support replaceability.

  Finally, the evaluation semantics used for contract assertions, item 4, and the maximum number of repetitions of evaluations, item 5, are both properties controlled by compiler build

[2] This addition could be argued as actually being remarkably small since the change was mostly in the form of wording clarifications in the specification for coroutines itself; no changes were made to the specification of Contracts other than removing the restriction against applying `pre` or `post` to a coroutine that had previously been in place.

flags. The C++ Standard has never — and likely will never — mandate specific build flags *because* each platform delivers them in unique ways and introduces new options based on user needs.

[P3321R0] offers a more detailed examination of what is expected of implementations for each of these items. That paper was presented to the tooling study group, SG15, at the Wrocław meeting; the group clearly indicated that the MVP's intent was understood, and no new concerns were raised that were not addressed during the discussion.

Importantly, the situations in which behavior has been made implementation defined are not cases for which we have been unable to find the correct solution, but well-known cases for which no single answer is a viable solution for every C++ program and platform. These situations are also not cases where 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.

- **No grouping of contracts** — The lack of syntactic control for contract assertions is certainly a concern, but the ability to introduce such things is a layer of complexity that has been explicitly left to future proposals that build on top of [P2900R13].

  Grouping of contracts with some form of syntactic labeling is far from a trivial design and comes with many competing needs. From the various use cases (as gathered in [P1995R1]) that would leverage such groupings to the ways in which semantics would need to be defined for such groupings (as described in [P2755R1]), the design space for this aspect of the feature is broad.

  From the start, a deliberate decision was made to delay such features until after an MVP is decided upon because those who have thought most about such features believe that consensus on a complete design will be achievable only *after* gaining more user experience with a Standard Contracts feature.

  [P3400R0], however, proposes a method that can, among other things, enable such groupings arbitrarily. Importantly, note the complexity needed for such a feature (albeit with a simple user interface) and recognize that the right design will take time and experience to gain consensus in WG21 and does not belong in an MVP.

- **Defaults** — For every aspect of the design where alternative options might be considered, the default behaviors in [P2900R13] have been selected based on a clear set of design principles and software engineering considerations.

- **Untried** — Large-scale use of [P2900R13] itself has not happened and cannot happen until it has shipped in production compilers, which requires first adding the feature to the C++ Standard. However, the basic aspects of the feature have been deployed in production contract-checking facilities *for decades*.

  – The model for preconditions and postconditions on virtual functions is compatible with all existing object-oriented design approaches and allows for the real use cases that occur in C++ code.

- No widespread experience exists with contract assertions that do not allow exceptions to escape from their predicate, mainly because many large codebases simply avoid the use of exceptions entirely, and even less frequently is a predicate used that might be potentially throwing.
- Extensive usage experience exists with linking translation units that use different "build modes" since this scenario is a primary case in which many enterprises accept a form of benign ODR violation to mix builds with different control macros (such as `NDEBUG` is for `assert()`).

- **Runtime vs. static checking** — The use of static analysis to check the correctness of a program aided by assertions is already in widespread use to varying levels of effectiveness, as explored in [P3386R0].

  The ability to specify predicates that cannot be executed is not part of the Contracts MVP, nor does it fit into the model where we are explicitly avoiding in-source control over contract-assertion behavior until we first lay the foundation for the feature with the MVP. The design for such things, however, has been shared for years — starting with [P2755R1] and now more thoroughly specified in [P3400R0]. This kind of functionality has been one of SG21's goals from the start, has been thoroughly explored, and is on our road map of features to provide over time.

- **Pointers to functions** — Expressing interest in features we might want is helpful to guide future expansion, yet we must understand that solutions for some problems might involve large efforts that will require extensive, careful design and consensus building. [P3327R0] offers a detailed exploration of the potential mechanisms that could be used to add `pre` and `post` to function pointers, and the paper was presented to EWG at Wrocław meeting. EWG, at the time, clearly understood that the scope and complexity of the problem indicate that no viable solution is ready for inclusion in the Contracts MVP.

  EWG, Wrocław, 2024-11-18

  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 result does not, however, mean that the topic of contracts on function pointers is abandoned. An aggressive effort is ongoing to introduce mechanisms to manipulate function contract assertions independently of an actual function, in the form of function usage types as described in [P3271R0] or [P3583R0] and will require time (and implementation experience) before all the details of a complete solution take shape. That said, this effort is one of the highest-priority goals of SG21 and is likely to evolve early in the C++29 timeframe.

  Note, very importantly, that functions invoked through function pointers in [P2900R13] still have their preconditions and postconditions evaluated. The only missing feature here is having a different set of contract assertions tied directly to a *pointer*, and significant work

and discussion indicates clearly that this topic should be addressed later. Similarly, when a programmer truly has an immediate need in the short term to wrap a function in such a way that different contract assertions are applied around its invocation, they can do so with [P2900R13] by specifying a lambda that forwards to the desired function (requiring no captures and thus being implicitly convertible to a function pointer) and instead specifies `pre` and `post` on the lambda expression.

- **Safety** — The only concrete proposal for Profiles that EWG is considering at this point is Herb Sutter's [P3081R1], which explicitly describes how it leverages [P2900R13] and requires no changes to [P2900R13] to do so. By design, any such runtime checking built on top of [P2900R13] is possible.

  Even more relevant for safety, the design specified in [P3100R1] is significantly broader in scope, is more widely applicable, and again builds on top of [P2900R13] without requiring any changes to the MVP (other than an option for improved ergonomics by adding enumerators to those enumerations provided by [P2900R13]). As described in [P3558R0], pursuing that direction for the language will not only benefit clients of new Standards, but will help improve the safety of C++ programs written against any Standard.

  The general question of how Profiles will interact with Contracts must be answered by the designers of Profiles. Given the ambiguity regarding what profiles should be able to do, what their scope is, and how they should be specified (e.g., the conflicting definitions in [P3589R0] compared to [P3081R1] and the concerns about Profiles raised in [P3543R0] and [P3586R0]), the only clarity at the moment is that the Profiles proposals have not yet addressed at all how they will provide usable and deployable runtime checking of any sort and that utilizing the work of SG21 (which can easily be leveraged) in that space is the only viable path forward.

- **Undefined behavior (UB)** — Requiring that the predicates of contract assertions be written in some UB-free dialect of C++ has been discussed many times over many years. SG21 has failed, three times, to reach consensus to enforce such a requirement.

  The primary proposals that attempt to address the perceived issues with undefined behavior when evaluating contract-assertion predicates are those that introduce conveyor functions — [P2680R1] and [P3285R0] — and do so to define some form of UB-free, side-effect–free subset of the language for contract assertions. These papers were discussed multiple times, and at no point was even a majority interested in pursuing these options or requiring that the Contracts MVP depend on them.

  – At the Kona meeting in November 2022, SG21 reviewed and polled [P3285R0] for the first time with no consensus to pursue it further.

  – After further clarifications on [P2680R1] to questions expressed in [P2700R0], a repeated discussion and a poll in a telecon in December 2022 once again failed to have consensus to pursue this design.

  – At the Tokyo meeting in March 2024, this concern was raised again, after which a joint in-person session of SG21 and SG23 once again discussed the issue and polled [P2680R1]. Strong consensus was again reached to avoid delaying the Contracts MVP until a "safe" programming model is available to which to restrict them.

– In May 2024, after a new paper proposing conveyor functions, [P3285R0], was produced, SG23 considered this proposal and attained strong consensus that conveyor functions were not required for the Contracts feature.

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

– During the Wrocław meeting in November 2024, another paper proposing the simplest starting point for "strict" predicates that have no UB, [P3499R0], was put forward, this time in EWG, and failed to achieve consensus for inclusion in the Contracts MVP.

Each discussion of a full specification concluded with a lack of certainty that such a specification could ever be produced that would succeed both at keeping the promises made by [P3285R0] while also being implementable and usable for nontrivial functions and contract assertions. In Section 4 of [P3386R0], each of the fundamental ideas in this proposal is examined, and the disappointing ramifications of each aspect of [P3285R0] are enumerated.

The overall question of undefined behavior — and of language safety in general — is much bigger than simply contract-assertion predicates and applies to all C++ code. Discussions within SG21 and SG23 have made eminently clear that solutions to bring language safety to C++ should not be contract specific. They can, however, leverage the Contracts feature to manage and mitigate undefined behavior in the same way that the Contracts feature can manage and mitigate library undefined behavior, and that solution is to continue to pursue [P3100R0]. Should we do so, the option to enforce implicit preconditions when evaluating contract predicates will be available and, again, fits completely within the foundation provided by [P2900R13]. This plan is explored more in [P3558R0], and in [**?**], we propose a concrete initial design for some of the most common sources of undefined-behavior-related errors in C++ programs.

- **Composition of TUs** — The potential designs for composing translation units with different configurations for contract assertions are explored thoroughly in [P3321R0]. These decisions have been discussed in SG15, with a clear understanding and acceptance of the intent for how the Contracts feature interacts with tooling in that paper.

  Regarding the other concerns listed in [P3573R0], the specific semantic chosen for a contract assertion are up to an implementation to explain, and valid reasons justify many different possible results (enumerated in [P3321R0]).

  For future users of C++, however, the *defined* aspect of *implementation defined* will be most significant. Each implementation will be documenting how to configure contract assertions when building a program, what factors effect the evaluation semantic that will be used, and how mixing different TUs with different configurations will take effect. Far better than the current situation (with macro-based solutions) of mixed modes leading to a program that

is IFNDR, mixed modes with [P2900R13] Contracts will instead reliably give us one of the configurations with which we have built a function. With future extensions to the full toolchain, these guarantees might even be extended, all within the space of conforming implementations defined by [P2900R13].

Functions that are `constexpr` or `consteval` — and constant evaluation of contract assertions in general — have been extensively discussed in the design of [P2900R13], and the model has been carefully crafted to meet our design principles; a more thorough explanation for that model can be found in [P2894R2]. The model primarily avoids giving different results in different contexts and instead makes any contract violation an error that is not subject to SFINAE.

The topic of Concepts has been discussed extensively as well, and the "Concepts Do Not See Contracts" principle follows from the prime directive in [P2900R13]. Not allowing the violation of a contract assertion to impact overload resolution or Concept satisfaction means that we avoid having multiple TUs with different understandings of a concept based on either the contract assertions that are present or the semantics with which they are evaluated.

- **Contracts and Modules** — The Contracts implementations in Clang and GCC are both part of compiler suites that also have basic support for Modules and, in some cases, are being developed by implementers who have been central to the implementation of Modules on those compilers.

  Basic testing with Modules has been performed with Contracts, and in general, the design of [P2900R13] gives the freedom needed for compiler vendors to provide the right integration to their users between selection of contract-assertion semantic and Modules. These alternatives are also enumerated in [P3321R0].

- **Contracts and static reflection** — Nothing in static reflection as proposed today ([P2996R9]) will allow inspection or reification of contract assertions. Accessing function contract assertions, or assertion statements, is not part of the proposal at this moment. Once both features are in the language, we expect to see growth in this direction, but that growth must also be carefully considered with respect to how it might be allowed without breaking the principles of [P2900R13].

- **Compatibility of future improvements** — Based on the nearly 200 use cases it has carefully gathered, SG21 has discussed and planned extensively for future improvements, and those plans have guided the design of [P2900R13] from the start. In particular, many of those plans are mentioned in [P2755R1], which includes potential mechanisms to address most (if not all) of the remaining concerns we are discussing here.

- **The proposal is far too large** — This concern is especially surprising given the desire expressed in [P3573R0] to support function pointers or grouping of contracts, both of which would make any contract-checking proposal significantly larger.

  More importantly, describing the basic facility — putting `pre`, `post`, and `contract_assert` into source code to improve its correctness — requires little time, and then the average user is prepared and ready to be productive. The rest of the proposal is highly detailed and thorough in describing *how* it interacts with our beautiful and complex language, but that size and

complexity is *in no way* a burden to users of the facility. Learning the more detailed nuances of interactions with the rest of the language will then happen over time as welcome error messages (instead of pernicious runtime failures) guide the developer to using the feature correctly. None of these rules places any sort of undue burden on the initial slope of the learning curve.[3]

Finally, the idea that contract checking is somehow simple and must therefore have a simple implementation is belied by the decades WG21 has invested in pursuing it to only now have reached the cusp of having a result available to users of C++, with both specifications and implementations. The paper trail for Contracts in WG21 is quite lengthy (going back to as early as [N1613] in March 2004), and a full documentation of that long path — and the many discussions and decisions that occurred along the way — can be found in [P2899R0].

Recognizing the needed core principles of Contracts as put forth in [P2900R13] and then integrating a design based on those principles into a vast and complex language such as C++ is not and could never be a simple task. On the other hand, the Contracts feature itself is orders of magnitude less complex and impactful on the language, as a whole, than similar features we have successfully standardized in the past, such as move semantics, `constexpr`, Concepts, Coroutines, and Modules.

- **Scalability** — The requirements of [P2900R13] put absolutely no expectations on linkers or generated ABIs related to contract assertions *because* [P2900R13] has been designed from the start with scalability in mind.

  We have actively designed against having tool-chain changes being forced, and [P3321R0] clearly states why that is not an issue.

The second set of concerns currently raised to WG21 were expressed in [P3506R0]. Many of that paper's concerns regarding the inadequacies of the Contracts MVP are duplicates of those expressed in [P3573R0] and are discussed above; only a few are distinct concerns.

- **Lack of deployment experience on most recent changes** — Lack of field experience with the proposed solutions for virtual functions and coroutines is claimed. While experimental compilers that support these features are available (and have been for quite a while), no large-scale projects using them have been undertaken. That real projects do not get designed from the ground up with the use of experimental features in mind is a simple, real-world fact.

- **Lack of experience with `pre` and `post`** — [P3506R0] states that the primary novelty of the Contracts MVP is putting function contract assertions on declarations and thus any effort to use an implementation of the Contracts MVP in existing code is irrelevant if its experiment is performed solely through the use of `contract_assert`.

  Many aspects of the feature — such as `const`-ification, the model for selecting evaluation semantics, and the model for handling exceptions — are all equally applicable to `contract_assert`. More importantly, the primary concern with the novelty of putting function contract assertions

[3] During the C++ London Meetup on January 20, 2025 ([londoncpp012025]), Timur Doumler was asked to describe [P2900R13], and he gave a complete synopsis of the usage of Contracts in just over two minutes. Another concise description has been published as a (small) blog post ([tdoumler2025]).

on declarations is that such assertions would then be exposed to client translation units. We do have, however, decades of experience exposing assertions to client translation units through the placement of assertions within inline functions in headers. The pros and cons of this approach are well known, and the MVP tackles the majority of the disadvantages by dint of not being based on the preprocessor.

3 Conclusion

When concerns are raised after a study group within WG21 has approved a paper to be forwarded toward the next stage, we must ask, as participants in the ISO process, whether the groups responsible for the decisions to forward the proposals to the next stage in the pipeline sufficiently discussed the concerns being raised. For all the concerns raised in [P3573R0], this paper has provided a brief summary showing that the concerns were indeed discussed thoroughly in all the relevant study groups — SG21, SG23, EWG, and LEWG. If the detail presented here is insufficient, [P2899R0] contains a much more thorough recording of the paper trail, discussion history, and polling that led to the Contracts MVP.

More to the point, the collection of concerns presented in [P3573R0] and [P3506R0] are so self-contradictory that the issues therein could never all be addressed. A Contracts proposal cannot be as small as possible yet contain all features that each user or small group of users might want. A proposal similarly cannot fully specify the behavior and avoid implementation-defined aspects while also supporting the scalability and usability concerns of the full spectrum of real-world C++ users.

We hope that this paper shows that the issues raised by [P3573R0] and [P3506R0] *have* been addressed and reassures those considering how to move forward with C++ that [P2900R13] is indeed both the right solution for adding contract support to C++ and is ready to be adopted for C++26.

Acknowledgments

Thanks to everyone who has contributed to developing [P2900R13] for adoption into C++26.

Thanks to John Lakos for early feedback and to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[tdoumler2025] Timur Doumler, “Contracts for C++ Explained in 5 Minutes”. https://timur.audio/contracts_explained_in_5_mins, 30 January 2025

[londoncpp012025] C++ London, “Contracts and Safety for C++26: An expert roundtable”. https://www.youtube.com/watch?v=NDyRiT3ZOMY&t=764s, 20 January 2025

[N1613] Thorsten Ottosen, “Proposal to add Design by Contract to C++”, 2004 http://wg21.link/N1613

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, “Contracts — Use Cases”, 2020 http://wg21.link/P1995R1

[P2680R1] Gabriel Dos Reis, “Contracts for C++: Prioritizing Safety”, 2023
http://wg21.link/P2680R1

[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

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, “A Bold Plan for a Complete Contracts Facility”, 2024
http://wg21.link/P2755R1

[P2811R7] Joshua Berne, “Contract-Violation Handlers”, 2023
http://wg21.link/P2811R7

[P2894R2] Timur Doumler, “Constant evaluation of Contracts”, 2024
http://wg21.link/P2894R2

[P2899R0] Timur Doumler, Joshua Berne, Andrzej Krzemieński, and Rostislav Khlebnikov, “Contracts for C++ - Rationale”, 2025
http://wg21.link/P2899R0

[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

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

[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

[P2996R9] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, and Dan Katz, “Reflection for C++26”, 2025
http://wg21.link/P2996R9

[P3071R1] Jens Maurer, “Protection against modifications in contracts”, 2023
http://wg21.link/P3071R1

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

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Contracts for C++: Support for virtual functions”, 2024
http://wg21.link/P3097R0

[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

[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

[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

[P3271R0] Lisa Lippincott, “Function Usage Types (Contracts for Function Pointers)”, 2024
http://wg21.link/P3271R0

[P3285R0] Gabriel Dos Reis, “Contracts: Protecting The Protector”, 2024
http://wg21.link/P3285R0

[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

[P3344R0] Joshua Berne, Timur Doumler, and Lisa Lippincott, “Virtual Functions on Contracts (EWG - Presentation for P3097)”, 2024
http://wg21.link/P3344R0

[P3386R0] Joshua Berne, “Static Analysis of Contracts with P2900”, 2024
http://wg21.link/P3386R0

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

[P3478R0] John Spicer, “Constification should not be part of the MVP”, 2024
http://wg21.link/P3478R0

[P3499R0] Lisa Lippincott, Timur Doumler, and Joshua Berne, "Exploring strict contract predicates", 2025
http://wg21.link/P3499R0

[P3506R0] Gabriel Dos Reis, "P2900 Is Still not Ready for C++26", 2025
http://wg21.link/P3506R0

[P3520R0] Timur Doumler, Joshua Berne, and Andrzej Krzemieński, "Wroclaw Technical Fixes to Contracts", 2024
http://wg21.link/P3520R0

[P3543R0] Mungo Gill, Corentin Jabot, John Lakos, Joshua Berne, and Timur Doumler, "Response to Core Safety Profiles (P3081)", 2024
http://wg21.link/P3543R0

[P3558R0] Joshua Berne and John Lakos, "Core Language Contracts By Default", 2025
http://wg21.link/P3558R0

[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

[P3583R0] Jonas Persson, "Contracts, Types & Functions", 2025
http://wg21.link/P3583R0

[P3586R0] Corentin Jabot, "The Plethora of Problems With Profiles", 2025
http://wg21.link/P3586R0

[P3589R0] Gabriel Dos Reis, "C++ Profiles: The Framework", 2025
http://wg21.link/P3589R0