Are Contracts "safe"?

Timur Doumler (papers@timur.audio)
Gašper Ažman (gasper.azman@gmail.com)
Joshua Berne (jberne4@bloomberg.net)
Ryan McDougall (mcdougall.ryan@gmail.com)

Document #: P3500R1
Date: 2025-02-09
Project: Programming Language C++
Audience: EWG

"The problem with contract assertions is not that they are not *safe*; the problem is that they are not *there*."

— Daisy Hollman, at an EWG session on P2900

"If we are serious about safety, we need to have Contracts."

— Peter Dimov, on the EWG mailing list

Abstract

The Contracts facility proposed in [P2900R13] and slated for inclusion in C++26 has sparked concerns that Contracts are not "safe". These concerns stem from two integral aspects of the proposed design. The first is that, while contract checks are *enforced* by default, they may optionally be *ignored* or *observed*, allowing execution to continue past a contract violation. The second is that contract predicates are boolean expressions that are evaluated according to the usual rules for C++ expression evaluation and thus can themselves exhibit undefined behaviour.

In this paper, we address the barriers to consensus on these issues. The main such barrier is the lack of a shared understanding of what we mean by "safety" and the failure to make the critical distinction between *language safety* and *functional safety*. We discuss this distinction, as well as the concepts of security and correctness, and clarify the purpose of Contracts with relation to those concepts, which is distinct from (and complementary to) features that focus on removing undefined behaviour from C++. We then describe the solutions available now and the extensions proposed, for the future evolution of Contracts, that address the stated concerns over [P2900R13]. Finally, we explain the critical importance, for the safety and security challenges facing the C++ ecosystem, of advancing — *not delaying* — the standardisation of the initial feature set proposed in [P2900R13] while we continue to develop such extensions.

1 The Concerns

The concerns that contract assertions as proposed in [P2900R13] are not "safe" stem from two integral properties of the proposed design. The first such property is that the model of flexible *evaluation semantics* provides the option to *ignore or observe* a contract check rather than *enforce* it, which in turn allows program flow to continue past a contract violation and into code that may have undefined behaviour:

```
T& MyVector::operator[] (size_t index)
pre (index < size()) {
  return _data[index];  // UB if index >= size()
}
```

The second property is that performing a contract check involves evaluating the contract predicate, which is a boolean expression and, therefore, follows the normal C++ rules for expression evaluation and can itself have undefined behaviour:

```
int f(int a) {
  return a + 100;
}

int g(int a)
  pre (f(a) > a);
```

In this example (originally from [P2680R1]), the compiler is allowed to assume that the signed integer addition inside `f` will never overflow and that, therefore, the precondition assertion of `g` will always evaluate to `true` and then elide the precondition check entirely, even if the evaluation semantic is *enforce* or *quick_enforce*.

The single greatest barrier to making any progress towards consensus is the lack of a shared understanding of what we mean by "safety" and the failure to make the critical distinction between *language safety* and *functional safety*. This lack of clarity, in turn, leads to a misunderstanding about how Contracts relate to the safety, security, and correctness of C++ code. We clarify these concepts in Section 2.

The second barrier is a lack of understanding of the *solution space*: how can the instances of undefined behaviour above be avoided while still having a Contracts facility that can be used effectively in practice? Various suggestions were made, such as the removal of *ignore* and *observe* semantics, the addition of an "always enforce" label, and the introduction of so-called "strict contracts". In Section 3, we explain why these suggestions are not workable and which actually viable solutions exist.

Finally, the third barrier is the idea that we could simply delay standardising Contracts beyond C++26 until we get consensus on a perfect solution that fully covers both language-safety and functional-safety needs. In Section 4, we discuss why, due to the safety and security challenges facing C++, we *must not delay* the standardisation of the initial feature set proposed in [P2900R13] as we continue to work on a broader solution.

2 Safety, Security, and Correctness

2.1 What do we mean by "safe"?

The single greatest barrier to progress is the current lack of shared coherent definition or understanding of the terms "safe" and "safety" in WG21. In the current debate, "safety" means different things to different people but is often used without further qualification, leading to misunderstanding and confusion (see also [P3578R0]).

In this paper, we never use the words "safe" or "safety" unqualified, except in quotes, and we strongly recommend that the rest of WG21 follows the same practice going forward. Instead, we distinguish the more precise terms *language safety*, *functional safety*, and *security*, which describe fundamentally different concepts (see [Sutter24]):

— **Language safety:** providing language-level guarantees that a program does not have unbounded undefined behaviour when executed,
— **Functional safety:** minimising the risk that the program could cause unintended harm to humans, property, or the environment,
— **Security:** enabling software to protect its assets from a malicious attacker.

The above definition of language safety matches the definition of "safety" given in [Abrahams23]) and the way the word "safety" is often colloquially used within WG21. Language safety is further subdivided into *memory safety*, *thread safety*, *type safety*, and so on. Memory-safe languages like Java also have a notion of "undefined behaviour" (e.g., in the case of a data race), but the set of possible behaviours is strictly bounded; for example, you may read stale, torn, or out-of-order values, but you cannot end up with a corrupted stack or a JVM crash. By contrast, undefined behaviour in C and C++ is unbounded. A slightly more nuanced definition of *language safety* is, therefore, that it provides invariants or limits on program behaviour in the face of bugs ([Carruth23]).

Language safety, while important, is *insufficient* to provide functional safety or security. Further, language safety is of a fundamentally different nature than the other two properties. A programming language is either language safe or not; a particular piece of code written in a *non*-language-safe programming language either does lead to undefined behaviour when executed, or it does not. This property can be reasoned about within a logical framework and enforced through language specification and tooling.

By contrast, *functional safety* is derived from the definition of “safety” that most of the world outside of WG21 uses (see also [P3578R0]). Functional safety is typically regulated by policy; the prerogative to decide if something is “safe” is typically reserved to governmental organisations and regulatory bodies ([P2026R0]). Functional safety and security are properties that are measured *statistically* — a program cannot be safe or secure in an absolute sense, just safe or secure "enough" for its intended purpose, which is typically established by certification processes. Notably, many programs written in C++ today are certified "safe" if they follow certain coding guidelines and engineering practices, regardless of C++ not being language safe.

2.2 The best code is *correct* code

Language safety, functional safety, and security are all tied together by a fourth concept central to the rest of this paper:

— **Correctness:** the property of a program to behave according to its specification, also called the *plain-language contract*. This specification may be explicit (e.g., described in documentation) or implicit (e.g., developer's intent, convention, common sense).

The specification and the developer's intent should normally never be to introduce undefined behaviour to the program or to ship a product that is not functionally safe or that has security vulnerabilities. Correct code is, therefore, by definition, simultaneously free from undefined behaviour, functionally safe, and secure.

The opposite is not true: a program can be free from undefined behaviour, functionally safe, and secure while still being *incorrect*, i.e., not behaving according to its specification. When developing a program for any particular purpose, *maximising correctness* should, therefore, be the ultimate goal of the developer; all other desirable properties follow from that.

Like functional safety, correctness is a statistical property that computers cannot reason about: in the general case, it is impossible to *prove* that a program is correct (the existence of formally provable algorithms notwithstanding).

2.3 The role of Contracts

Contracts, as proposed in [P2900R13], are, first and foremost, *correctness assertions*. The importance of Contracts for C++ is that it is the *only* language feature currently proposed for C++ that explicitly targets correctness. The very purpose of Contracts is to give the user a tool to describe a subset of the plain-language contract — conditions under which the program is considered incorrect — in code and to make those conditions checkable during program execution. As discussed above, that *plain-language contract* of the program is typically unknowable to the computer and thus must be manually injected by a human via appropriate contract assertions. This is a key difference to language-safety properties, which can be described formally and for which checks can be generated in an automated fashion.

A Contracts facility is, therefore, undoubtedly a feature that improves *functional* safety. Often, the root causes of functional-safety issues are not manifestations of undefined behaviour, but *logical bugs* that could happen in any programming language. Consider, for example, a medical device malfunctioning because the developer committed an off-by-one error, a self-guided missile hitting the wrong target because the developer failed to account for accumulating floating-point errors, or a spacecraft veering off course and being destroyed because the wrong units were used (as famously happened with the Mars Climate Orbiter). All these examples are functional-safety failures; adding language-safety guarantees would do nothing to prevent bugs of this kind. In some cases, they can be prevented by leveraging the C++ type system (a user-defined floating-point type with built-in compensated summation; a strongly typed units library; and so on); in cases where such compile-time techniques are not applicable, adding contract assertions to your code is the tool of choice to detect the bug.

Similarly, a Contracts facility is undoubtedly a security feature. Language safety is necessary but insufficient for security. In fact, most of the largest data breaches and other cyberattacks in recent years were not related to language safety at all; consider Log4Shell, SQL injection, and others. Adding language-safety guarantees would do nothing to prevent vulnerabilities of this kind because they are not manifestations of undefined behaviour, but flaws in design. Contract assertions can both expose flaws in design and identify problems resulting from those flaws during execution. No language-safety feature can achieve this because such features are inherently limited to checks of low-level properties, such as "is this pointer valid" that can be generated and inserted automatically. On the other hand, contract assertions allow the developer to add *manual* checks for higher-level aspects of the design that represent the specification and the developer's intent and that cannot be formally expressed in a language specification or reasoned about by a C++ compiler.

The range of program defects that can be identified via contract checks includes program states that would lead to undefined behaviour if control flow were to continue. Thus, Contracts can improve language safety as well. However, they are not primarily intended to be a language-safety feature, and interpreting them as a language-safety feature is a conceptual fallacy. Contract checks introspect program states, but they do not add guarantees that remove undefined behaviour from the language and, in fact, do not change the semantics of the language at all. This is by design: by adding contract assertions to their code, programmers can find bugs that fell through the cracks of those language guarantees. Contracts are, therefore, *complementary* to proposals that add language-safety guarantees.

3 The Solution Space

3.1 Undefined behaviour following a nonenforced check

Enforcing a contract assertion is the safe default and often the correct choice; the *enforce* semantic is the recommended default in [P2900R13] for good reason. The *enforce* semantic is appropriate during development and, in some contexts, also in production, in particular when good runtime diagnostics are required. The other enforcing semantic, *quick-enforce*, is useful when language safety is the highest priority and terminating fast and rebooting is an acceptable mitigation strategy. *Quick-enforce* is a common strategy in embedded systems; library hardening ([P3471R0]) is a noteworthy example of using *quick-enforce* for generic code. Both enforcing semantics are "safe" in the sense that they disallow program flow to continue past an enforced contract violation, and instead the program is always terminated as part of evaluating a violated contract assertion. In the first example in Section 1, enforcing the precondition assertion on `operator[]` thus means that function will never have undefined behaviour.

At the same time, unconditional termination on contract violation is completely unacceptable (see [P2698R0]) in some contexts. In those situations, e.g., video games, not crashing the program is much more critical to commercial success than preventing certain kinds of bugs from manifesting. In other contexts, such as large server systems, one might want to add *new* contract assertions to an *old* existing system that is known to run successfully in

production without risking bringing down the entire system if the assertion itself turns out to be overly strict or otherwise written incorrectly. In yet other contexts, such as ultra-low-latency applications and high performance computing, the runtime overhead of performing the contract check might be prohibitive in production and affordable only during development.

Reusable, generic code normally does not know about the context in which it will be used and thus cannot make the choice of evaluation semantic. Therefore, every existing Contracts facility — be it a language feature, such as in Eiffel, D, or Ada, or a library feature, such as C `assert` or custom assertion macros — provides the ability to turn off the checks. This property — that the checks are *redundant*, do nothing in a correct program, and can be turned off if desired — is the key difference between assertions and control-flow features, such as `if`-statements ([Lippincott24]). Removing this property would make contract assertions unsuitable and undeployable in a vast number of scenarios.

If we want Contracts to succeed, we, therefore, cannot remove the nonenforcing semantics. A suggested alternative, for those safety-critical use cases that require a guarantee that a contract assertion will always be enforced, is to add an always-terminate label:

```
T& MyVector::operator[] (size_t index)
  pre [[always_terminate]] (index < size());
```

Such a label would constrain the possible choice of semantic to *enforce* or *quick-enforce* (or possibly, *quick-enforce* only), thereby providing a hard guarantee that control flow will never continue into the body of the function if the postcondition has been violated, regardless of how the build has been configured.

While we should certainly aim to provide this functionality for C++29, it is unfortunately out of scope for C++26 for two reasons.

The first reason is technical. While an always-terminate label is an important use case for a label that constrains the possible semantics of a contract assertion, it is not the *only* use case. Other important use cases include a never-terminate label to enable scenarios where termination is not acceptable, an always-apply-this-exact-semantic label to enable unit testing of contract assertions, and a never-apply-a-checking-semantic label to encode contract assertions that are intended for static analysis only and that cannot or should not be evaluated at run time for whatever reason.

Thus, providing an always-terminate label on its own would compromise the Contracts design similarly to a hypothetical special syntax for instantiating a template with `int` as the template argument: while `int` is arguably the most common type in C++, special-casing C++ template syntax for that particular type is not a reasonable language design.

The second reason is nontechnical. The last time EWG decided to add new syntax to a Contracts proposal shortly before the feature freeze deadline for a C++ Standard in order to remove a sustained objection — [P1607R1] at the July 2019 meeting in Cologne — it ultimately led to removal of the entire proposal from the C++ Working Draft ([P1823R0]).

Given how the internal dynamics of WG21 work, there is a real risk that the same thing would happen this time around.

The correct solution to address the problem is a properly designed facility for specifying labels on a contract assertion that constrain the possible evaluation semantics of that assertion. Such labels could be defined by the user as well as provided by the C++ Standard itself; an `always_terminating` label should certainly be one of the labels provided by the C++ Standard.

The good news is that [P2900R13] has been explicitly designed from the start with this extension in mind, and a proposal for this extension is available in [P3400R0]. With this proposal, an always-terminating contract assertion could be written as follows:

```
T& MyVector::operator[] (size_t index)
  pre <always_terminating> (index < size());
```

where `always_terminating` is an *assertion-control object* provided by the Standard Library and defined as follows:

```
namespace std::contracts {
  struct always_terminating_t {
    // ...
    consteval evaluation_semantic compute_semantic(evaluation_semantic input) {
      if (input == evaluation_semantic::quick_enforce) {
        return input;
      }
      else {
        return evaluation_semantic::enforce;
      }
    }
  }
}

namespace std::contracts::labels {
  constexpr always_terminating_t always_terminating;
}
```

We are confident that this extension will address all the above concerns regarding nonenforced contract assertions. Nevertheless, given the amount of time required, procedurally, to move even a well-crafted proposal through all relevant subgroups of WG21, including plenary, this extension is realistically too nontrivial to be approved in the C++26 timeframe and will have to target C++29. At the same time, delaying the base proposal, [P2900R13], until that extension becomes available (see Section 4) would be a critical mistake.

While we wait for an extension that will add semantic-constraining labels to Contracts, a number of workarounds to achieve always-terminating contract assertions are available right now. First of all, the Clang implementation offers a semantic-constraining label as a

vendor-specific attribute, and we expect the GCC implementation to provide an analogous attribute in due course:

```
T& MyVector::operator[] (size_t index)
  pre [[clang::contract_semantic("quick_enforce")]] (index < size());
```

When working with a compiler that does not provide such a vendor-specific attribute, we can implement such always-terminating checks with an `if`-statement, which is what developers currently do:

```
T& MyVector::operator[] (size_t index) {
  if (index >= size())
    std::abort();  // or __builtin_trap or some other strategy

  return _data[index];
}
```

This strategy does have the disadvantage that contract violations will not call the global contract-violation handler, but such a call can be achieved by duplicating the check:

```
T& MyVector::operator[] (size_t index)
  pre (index < size()) {
  if (index >= size())
    std::abort();  // or __builtin_trap or some other strategy

  return _data[index];
}
```

If advertising the precondition on the function interface is not required, the duplication can be removed by wrapping it into a macro:

```
#define ALWAYS_TERMINATING_CONTRACT_CHECK(x) \
  contract_assert(x); if (!x) std::abort();
```

Further, if [P3290R2] is adopted into C++26 (it has already been approved by SG21 but not yet seen by LEWG at the time of writing), the above assertion can be written with no duplication by calling the contract-violation handler explicitly as follows:

```
T& MyVector::operator[] (size_t index) {
  if (index >= size())
    std::contracts::handle_enforced_contract_violation("Out of bounds!");

  return _data[index];
}
```

This call can again be wrapped into a macro to save some typing if desired.

Finally, an organisation wishing to disallow nonterminating contract assertions could always enforce the appropriate compiler options for selecting terminating semantics in their build

chain. The Clang implementation of [P2900R13] offers a `-fcontract-evaluation-semantic` flag for selecting the semantic per translation unit, and the GCC implementation intends to follow suit and harmonise their semantic-controlling flags with those implemented by Clang. We do not know how other compilers will eventually implement the selection of contract-evaluation semantics and what options they will provide, but we are confident that user-friendly selection mechanisms that play well with existing build systems will be readily available.

3.2 Undefined behaviour during predicate evaluation

The second concern is that contract checks are not langage safe because a contract check involves the evaluation of the predicate, which is a boolean expression, and that evaluation is in itself not language safe because various kinds of undefined behaviour can occur when evaluating an expression in C++ (e.g., dereferencing a valid pointer, integer overflow, and so on). This concern has been brought up repeatedly (see [P2680R1], [P3173R0], [P3285R0], and [P3362R0]); the solution suggested in these papers is to restrict the expressions allowed in the contract predicate to those that can be statically proven to be free from undefined behaviour (excluding race conditions).

The discussions around this idea of *strict contracts* have been ongoing for almost three years now; SG21, SG23, and EWG all had multiple sessions on this topic. Each time the question was polled, the respective group has decided in favour of the direction proposed in [P2900R13]; the poll results are documented in [P2899R0]. An important factor is that, unfortunately, the proponents of strict contracts have not produced a complete specification, and whether the approach is indeed specifiable and implementable appears doubtful. Deeper analysis (see [P3376R0] and [P3386R0]) revealed that strict contracts, based on the principles that can be gleaned from [P3285R0], result in only a tiny number of predicates that would be viable to express, with a huge amount of new language complexity needed to achieve anything beyond the most basic arithmetic operations.

In particular, we do not yet know whether or how one could use pointers and references to objects or call any member function of an object in a strict contract predicate since as we are unaware of any technique applicable to C++ that could statically prove that a pointer or reference points to a valid object. It seems possible and even likely that no approach using *local* static analysis, such as the `std::object_address` sketch provided in [P3285R0], could provide such a proof and that only the introduction of *global* language constraints on the existence of mutable references to objects can achieve this, such as a borrow checker ([P3390R0]), mutable value semantics ([Racordon2022]), or outlawing mutation of objects altogether (as in pure functional languages); see also [Baxter2024].

In [P3499R0], we explored a design for strict contracts that is actionable — i.e., specifiable and implementable. Unfortunately, it is also severely limited. While the strict contracts described in that paper are guaranteed to be free of undefined behaviour when checked, they do not allow any operation on values other than of built-in arithmetic or enumeration type; dereferencing any pointers; using any references to objects; or calling any existing member function on any object. For example, we could not even check the size of a `std::vector` in a strict predicate or whether it is empty, not even if that `std::vector` is

passed in by value, because we cannot construct a proof that the `this` pointer is valid. Strict predicates seem, therefore, to be of no practical use for the vast majority of contract assertions one might want to add to a real-world C++ codebase.

Overall, we are not opposed to exploring the idea of strict predicates to see if the set of permissible expressions can be expanded further, but delaying [P2900R13] until this work is complete would be an egregious disservice to the C++ community (see Section 4). A much more promising approach is to promote efforts to remove undefined behaviour that are applicable to the *entire* C++ language, such as the framework proposed in [P3100R1]. Once we have those tools in place for evaluating C++ expressions in general, we can seamlessly apply them to contract predicates as well, thus removing the concern without reducing the utility of the Contracts facility proposed in [P2900R13].

4 Why we need Contracts in C++26

Insufficient functional safety has a high cost, sometimes including human lives. As members of WG21, we have a moral responsibility to prioritise functional safety when evolving the C++ language. Focusing exclusively on increasing *language safety* — i.e., the absence of undefined behaviour — is not an effective approach because, as we saw in Section 2, many functional-safety issues are not manifestations of undefined behaviour and are thus unaided by efforts to provide language-safety guarantees but can be effectively addressed by adding contract assertions. Failing to ship Contracts in C++26 is, therefore, actively harmful to efforts to improve functional safety in C++ (see also [P3297R1]).

Similarly, addressing security in C++ is an urgent problem. However, focusing exclusively on increasing language safety to address security is not an effective approach since it overlooks a huge chunk of the problem. As discussed in more detail in [Sutter24], the oft-quoted 70% of CVEs caused by the lack of language memory safety ([Gaynor20]) is misleading: the consensus among security experts is that even if we could somehow make C++ a memory-safe language, that would *not* lead to 70% fewer CVEs, data breaches, and ransomware attacks ([Hanley24]). Addressing security issues that are *not* manifestations of undefined behaviour but are caused by other engineering flaws in the program is thus equally important. In many cases, such flaws can be identified by the appropriate use of contract assertions.

Today, the use of contract assertions *at scale* to identify such flaws is hindered because the tools that we have currently in C++ to write them — i.e., assertion macros — are too limited. Assertion macros cannot be placed on declarations, limiting their visibility to compilers, IDEs, and static analysis tools; in addition, they suffer from a number of other disadvantages due to being macros as opposed to core-language features (see [P2899R0], Section 2.1 "What Are Contracts For?" for a more thorough discussion). Further, the C `assert` macro suffers from a lack of configurability: it offers a diagnostic message followed by program termination as the only mitigation strategy, which severely limits its applicability. This limitation can be overcome by custom assertion macros, but they do not have a standard syntax and semantics and thus cannot be used portably and scalably across different components of a large C++ program.

[P2900R13] is designed to remove all these limitations, to have the widest possible range of applicability, and to minimise impediments to its adoption in any C++ codebase of any coding style and any quality. Contracts can be leveraged to improve correctness incrementally in a way that no other proposed feature can do. The addition of a single contract assertion into an existing C++ application will help improve the program's correctness and stability, and each new assertion will add layers to this benefit.

The primary goal of this design is to have *more correctness checks in more places* across the entire C++ ecosystem, which will, in turn, make C++ code both more functionally safe and more secure, in ways that language safety by itself will never be able to achieve. Failing to ship Contracts in C++26 would significantly impede accomplishing this goal.

Once we have adopted [P2900R13] for C++26 as a solid foundational Contracts facility, we can then pursue a labels framework along the lines of [P3400R0], as an extension for C++29, as well as invest more effort into exploring the feasibility of strict contracts ([P3499R0]) if continued interest supports it. While valuable, none of these directions are necessary for Contracts to achieve their primary goal of providing more correctness checks in more places, across the entire C++ ecosystem, and thus such extensions should not block the adoption of [P2900R13].

[P2900R13] is intended to be a Contracts MVP (Minimal Viable Product) — a foundation upon which we can build, rather than a feature that addresses every possible use case in the first version that we ship. Such incremental standardisation is a tried and tested approach that we should not abandon. Consider the first iteration of `constexpr` functions in C++11. It was extremely limited and allowed only a single return statement inside the body of a `constexpr` function. Subsequently, C++14 added more functionality, C++17 yet more, and C++20 yet more. If we had required that `constexpr` satisfy every important use case in the first version of the feature that we shipped, we would not have `constexpr` in C++ today. Not addressing every possible use case in the first version of a Contracts feature should, therefore, not be a reason to delay shipping such a first version in a C++ Standard when it is ready.

Delaying [P2900R13] beyond C++26 because the proposed Contracts MVP does not provide language-safety guarantees that it was never designed to provide, aiming at a problem that it was never designed to solve, would send an alarming signal to the C++ community that, as a response to political pressure, WG21 has adopted a myopic focus on language safety and no longer cares about the fundamental value of facilitating *correct* code nor about security nor functional safety — the cost of which is measured in human lives — except in those areas where they intersect with language safety. Taking such a direction for the evolution of the C++ language would be irresponsible and unethical. Instead, we must recognise that language-safety guarantees and Contracts are *complementary* and that we need *both*.

Delaying [P2900R13] beyond C++26 would also pose a number of nontechnical risks. Proposals to add Contracts to C++ have been in development for over twenty years. Three previous attempts to standardise such a proposal ([N1613], [N4378], and [P0542R5]) failed. [P2900R13] is the culmination of the *fourth* such iteration. After such a long time and so

many failed attempts, continued disagreement within the committee would foster the impression that the committee is simply incapable of producing a Contracts feature.

In addition, the people still involved in developing this feature will grow increasingly frustrated and struggle to justify, to themselves and perhaps to their employers, spending more of their time and effort on a feature that seems doomed to disagreement. If Contracts are not adopted into C++26, the likely result is, therefore, not Contracts in C++29, but no Contracts in C++ at all.

The level of detail with which every possible aspect of the design proposed in [P2900R13] has been carefully considered, explored, and documented (see [P2899R0]) is unprecedented in WG21. The proposal is mature, its design is stable, it has been implemented in two major compilers (see [P3460R0]), it has received strong consensus in SG21 (a particularly challenging study group with the highest possible bar on consensus), and it has been approved by both EWG and LEWG for C++26, with a plenary vote expected at the upcoming WG21 meeting in February 2025 in Hagenberg. Contracts are ready for C++26 and are urgently needed to address the security and functional-safety challenges that the C++ ecosystem is facing.

Acknowledgements

Thanks to Greg Marr and Andrzej Krzemieński for reviewing this paper and providing technical feedback. Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Revision History

R0 → R1

- Fixed errors in Section 3.1 code examples
- Expanded discussion of different definitions of "safety" in Section 2.1
- Editorial changes throughout the paper

Bibliography

[Abrahams23] Dave Abrahams: "Value Semantics: Safety, Independence, Projection, & Future of Programming". CppCon 2022

[Baxter2024] Sean Baxter: "Why Safety Profiles Failed". 2024-10-24

[Carruth23] Chandler Carruth: "Carbon Language Successor Strategy: From C++ Interop to Memory Safety". CppNow 2023

[Gaynor20] Alex Gaynor: "What science can tell us about C and C++'s security". 2020-05-27

[Hanley24] Zach Hanley: "Rust Won't Save Us: An Analysis of 2023's Known Exploited Vulnerabilities". 2024-02-06

[Lippincott24] Lisa Lippincott" "Perspectives on Contracts for C++". CppCon 2024

[N1613] Thorsten Ottosen: "Proposal to add Design by Contract to C++". 2004-03-29

[N4378] John Lakos, Nathan Myers, Alexei Zakharov, and Alexander Beels: "Language Support for Contract Assertions (Revision 10)". 2015-02-08

[P0542R5] Gabriel Dos Reis, J. Daniel García, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup: "Support for contract based programming in C++". 2018-06-08

[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall: "Minimizing Contracts". 2019-07-23

[P1823R0] Nicolai Josuttis, Ville Voutilainen, Roger Orr, Daveed Vandevoorde, John Spicer, and Christopher Di Bella: "Remove Contracts from C++20". 2019-07-18

[P2026R0] Ryan McDougall, Bryce Adelstein Lelbach, JF Bastien, Andreas Weis, Ruslan Arutyunyan, Ilya Burylov, Glenn Elliott, Clint Liddick, Philipp Robbel, Aditya Sreekumar, Vasumathi Raman, and Jake Askeland: "A Constituent Study Group for Safety-Critical Applications". 2020-01-14

[P2680R1] Gabriel Dos Reis: "Contracts for C++: Prioritizing Safety". 2022-12-15

[P2698R0] Bjarne Stroustrup: "Unconditional termination is a serious problem". 2022-11-18

[P2899R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++ — Rationale". 2024-12-17

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-01-13

[P3173R0] Gabriel Dos Reis: "P2900R6 May Be Minimal, but It Is Not Viable". 2024-02-15

[P3290R2] Joshua Berne, Timur Doumler, and John Lakos: "Integrating Existing Assertions with Contracts'. 2024-09-06

[P3297R1] Christian Eltzschig, Mathias Kraus, Ryan McDougall, and Pez Zarifian: "C++26 Needs Contract Checking". 2024-06-21

[P3285R0] Gabriel Dos Reis: "Contracts: Protecting The Protector". 2024-05-15

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

[P3376R0] Andrzej Krzemieński: "Contract assertions versus static analysis and 'safety'". 2024-10-14

[P3386R0] Joshua Berne: "Static Analysis of Contracts with P2900". 2024-10-15

[P3400R0] Joshua Berne: "Specifying Contract Assertion Properties with Labels." 2024-12-17

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

[P3471R0] Konstantin Varlamov and Louis Dionne: "Standard library hardening". 2024-10-15

[P3499R0] Timur Doumler, Lisa Lippincott, and Joshua Berne: "Exploring strict contract predicates". 2025-01-13

[P3578R0] Ryan McDougall: "What is Safety?". 2024-12-12

[Racordon2022] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta: "Implementation Strategies for Mutable Value Semantics". Journal of Object Technology. Vol. 21, No. 2, 2022

[Sutter24] Herb Sutter: "C++ Safety in Context". 2024-03-11