Are Contracts "safe"?

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

Document #: P3500R0
Date: 2025-01-13
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 the February 2024 WG21 Meeting in Tokyo

"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* on the one hand and *functional safety* on the other hand. 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 argue why it is critically important for the safety and security challenges facing C++ that we *do not delay* the standardisation of the initial feature set proposed in [P2900R13] as we continue to work on 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 *enforcing* 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. As such, it follows the normal C++ rules for expression evaluation and thereby can itself have undefined behaviour. Consider:

```
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, 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* on the one hand and *functional safety* on the other hand. This, in turn, leads to a misunderstanding of 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 it is critically important for the safety and security challenges facing C++ that we *do not delay* the standardisation of the initial feature set proposed in [P2900R13] as we continue to work on such a 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 very 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 different concepts (see [Sutter24]):

- **Language safety:** providing language-level guarantees that a program does not have unbounded undefined behaviour when executed (this is further subdivided into *memory safety*, *thread safety*, *type safety*, etc).
- **Functional safety:** reducing the risk that the program could cause unintended harm to humans, property, or the environment;
- **Security:** making software able to protect its assets from a malicious attacker.

A key realisation is that language safety, while important, is *not sufficient* to provide software safety or security. Another key realisation is that language safety is of fundamentally different nature from the other two properties. A programming language, or program written in that language, is either language-safe or not; this property can be reasoned about within a logical framework and enforced through language specification and tooling (static analysers, sanitisers and so on). On the other hand, functional safety and security are *statistical* properties that computers cannot reason about and that are typically enforced by regulation.

All three concepts are 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 has security vulnerabilities. Correct code is therefore by definition simultaneously language-safe, functionally safe, and secure. The opposite is not true: a program can be language-safe, 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).

2.2 The role of Contracts

Contracts as proposed in [P2900R13] are, first and foremost, *correctness assertions*. The importance of Contracts for C++ is that they are 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 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 safety feature with regard to 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 has failed to account for accumulating floating point errors, or a spacecraft crashing because of the wrong units being used (as famously happened with the Mars Climate Orbiter). All of these are software safety failures; however, 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; etc); in other cases where that approach is not possible, adding contract assertions to your code is the tool of choice to detect them.

Similarly, a Contracts facility is undoubtedly a security feature. Language safety is necessary for security, but it is not sufficient. 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, etc. 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.

While the range of program defects that can be identified via contract checks includes instances of undefined behaviour, and thus Contracts can improve language safety as well, they are not primarily intended to be a language-safety feature, and interpreting them as a language-safety feature is a conceptual fallacy. Contracts 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, the programmer 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 non-enforced 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. It 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. Library hardening ([P3471R0]) is a noteworthy example of using *quick-enforce*; it is also a common strategy in embedded systems. Both enforcing semantics are "safe" in the sense that they have the property that program flow will never continue past an enforced contract violation, 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 that function will never have undefined behaviour.

At the same time, there are contexts where unconditional termination on contract violation is completely unacceptable (see [P2698R0]). In some contexts such as 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 only affordable 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 `cassert` or custom assertion macros – provides the ability to turn the checks off. 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 cannot remove the non-enforcing semantics. As an alternative, it has been suggested that, for those safety-critical use cases where a guarantee that a contract assertion will always be enforced is required, we could add an always-terminate label along the lines of

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

that 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 a number of 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 which cannot or should not be evaluated at runtime 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 non-technical. 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 of the above concerns regarding non-enforced contract assertions. Nevertheless, given how long it takes procedurally to move even a very well-crafted proposal through all relevant subgroups of WG21 including plenary, it is realistically too non-trivial to be approved in the C++26 timeframe; it will have to be an extension targeting C++29. At the same time, it would be a fatal mistake to delay the base proposal, [P2900R13], until that extension becomes available (see Section 4).

While we are waiting 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, it is always possible to do what people already do today to implement such always-terminating checks – to use an `if`-statement:

```
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. This can be achieved by duplicating the check as follows:

```
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 as follows:

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

Further, if [P3290R2] makes it 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 without any duplication as follows:

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

  return _data[index];
}
```

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

Finally, an organisation wishing to disallow non-terminating 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 essentially that contract checks are not langage-safe because they evaluate a boolean expression (the predicate) and the evaluation of a boolean expression in C++ is itself not language-safe. This concern has been brought up repeatedly (see [P2680R1], [P3173R0], [P3285R0], [P3362R0]), and the discussions around it have now been ongoing for almost three years, including multiple sessions in all of SG21, SG23, and EWG. 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].

A relevant factor in this discussion is that unfortunately, the proponents of strict contracts have not produced a complete specification, and it appears doubtful whether the approach is indeed specifiable and implementable. Deeper analysis of the idea in [P3376R0] and [P3386R0] revealed that strict contracts, based on the principles that can be gleaned from [P3285R0], result in only a very small number of predicates that would be viable to express, with a huge amount of new language complexity needed to achieve anything beyond the most basic arithmetic operations. In particular, we do not yet know whether or how one could use pointers and references to objects or call any member function of an object in a strict contract predicate, as we are not aware 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 not have undefined behaviour when checked, they also 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. It seems therefore that strict predicates are 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 further to see if the set of permissible expressions can be expanded further, but delaying the rest of [P2900R13] until this work is complete makes no sense and would do a massive 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 software safety has a high cost, and sometimes that cost includes human lives. As members of WG21 we thus have a moral responsibility to prioritise software 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 software safety issues are not manifestations of undefined behaviour and are thus not helped 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 in order to address security is not an effective approach either as it leaves out 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]). It is thus equally important to address security issues that

are *not* manifestations of undefined behaviour, but are caused by other engineering flaws in the program. 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 are hindered by the fact that the tools we have in C++ today to write them – 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 only 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. This in turn will 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 do significant harm to 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 efforts into exploring the feasibility of strict contracts ([P3499R0]) if there is continued interest in doing so. While valuable, none of these directions are necessary for Contracts to achieve their primary goal – more correctness checks in more places, across the entire C++ ecosystem – and thus they should not block the adoption of [P2900R13].

[P2900R13] is intended to be a Contracts MVP (Minimal Viable Product) – a foundation that we can build upon, 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: it 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` satisfies every important use case in the first version of the feature that we ship, we would not have `constexpr` in C++ today.

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 a fatal signal to the C++

community: that as a response to political pressure, WG21 has adopted a myopic focus on language safety and we no longer care about the fundamental value of facilitating *correct* code, nor do we care about security or functional safety – the cost of which is measured in human lives – except in those areas where it intersects 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 another risk. The more time passes without a Contracts facility being adopted into the C++ Standard, the harder it becomes to justify spending time, effort, and money on the continuing development of this feature. Realistically, if Contracts do not make C++26, it is highly likely that the people and companies currently investing in this feature will simply withdraw. The result of that would be *not* Contracts in C++29, but no Contracts in C++ at all.

Proposals to add Contracts to C++ have been in development for over twenty years. [P2900R13] is the culmination of the *fourth* iteration of this effort (after [N1613], [N4378], and [P0542R5]), and has itself been in development for over five years. The level of detail with which every possible aspect of its design 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 they are urgently needed to address the challenges that the C++ ecosystem is facing in these times.