Undefined and erroneous behaviour is a contract violation

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

Document #:	P3100R1
Date:	2024-10-16
Project:	Programming Language C++
Audience:	EWG, SG21, SG23

Abstract

In this paper, we propose to apply the semantics of Contracts [P2900R9] to undefined behaviour in C++. By respecifying evaluations of core language constructs that would have undefined behaviour as *contract violations*, we allow them to be checked, observed, enforced, and handled via a user-defined contract-violation handler, just like violations of user-authored contract assertions. We thus add a powerful and flexible new tool for improving the safety of C++. We also create a framework for reasoning, within the C++ Standard, about sanitisers and other existing strategies for mitigating undefined behaviour during program execution. Our proposed specification supersedes the concept of *erroneous behaviour* introduced by [P2795R5].

Contents

1 Introduction . . . 2
2 The role of Contracts for Safe C++ . . . 3
3 Defining undefined behaviour . . . 4
  3.1 Library UB vs. Core UB . . . 4
  3.2 UB vs. EB . . . 6
  3.3 UB with vs. without safe fallback behaviour . . . 7
  3.4 UB that is easy vs. difficult to diagnose during program execution . . . 9
  3.5 Explicit vs. implicit UB . . . 9
4 Adding the missing pieces . . . 10
  4.1 The *assume* semantic . . . 10
  4.2 Library API extension . . . 10
5 Discussion . . . 11
  5.1 Contracts provide a framework for safety . . . 11
  5.2 Contracts provide a program-wide handler . . . 12
  5.3 Contracts provide better integration for sanitisers . . . 12
6 Proposal summary . . . 13

1 Introduction

Improving safety and security is arguably the most important challenge for the future evolution of the C++ language ([Bastien2023]). Governments and regulatory bodies have called for the tech industry to move away from C and C++ and towards *safe* languages ([NSA2022], [CR2023], [CISA2023]). Rust is a frequently cited example of a language that is considered *safe*.

According to the definition in [Carruth2023], safety is characterised by invariants or limits on program behaviour in the face of bugs; safety bugs are bugs where some aspect of program behaviour has no invariants or limits. According to the definition of [Abrahams2023], a safe operation is one that cannot cause undefined behaviour; a safe language has only safe operations.

With the above definitions, C++ is not a safe language: both the core language and the C++ standard library allow for a multitude of ways in which a well-formed C++ program can exhibit *unbounded* undefined behaviour, i.e. situations in which the C++ standard places no restrictions on the behaviour of the program (see [P1705R1] for a list). Bugs that trigger such unbounded undefined behaviour cause stability issues and can be exploited by threat actors, thereby causing security vulnerabilities. Among the most popular programming languages, the presence of unbounded undefined behaviour is a unique trait of C and C++.

Realistically, we do not believe that it is possible to evolve C++ into a safe language under the above definition — i.e., remove all undefined behaviour from C++ — in the foreseeable future ([Doumler2023]). However, we can and should do what we can to give people significantly more control over the amount of undefined behaviour in C++ programs, especially with the aim of vastly *reducing* it, including in legacy code that may be recompiled but not modified on the source level.

Many ways to detect and mitigate undefined behaviour in C++ exist today. They range from engineering practices such as code reviews, coding guidelines, and meticulous coverage with unit tests to specialised tooling such as static analysers and sanitisers. However, while the use of such tools is usually strongly encouraged, they are not part of the core language and are therefore opt-in rather than opt-out. This is a major weakness of C++. For this reason, it is of particular importance to devote special attention to efforts to remove undefined behaviour from C++ on the language specification level.

These efforts are ongoing. For C++23, a common cause of undefined behaviour in range-based `for` loops has been removed ([P2012R2]). For C++26, it will no longer be undefined behaviour to execute a trivial infinite loop ([P2809R3]) and, in some cases,[1] to read an uninitialised value ([P2795R5]). Efforts to tame undefined behaviour at a larger scale include the current work on safety profiles ([P3274R0]) and adding a Rust-like borrow checker to C++ ([P3390R0]). Outside of the C++ Standard committee, solutions in this space are being shipped by compiler vendors, for example the hardening modes in libc++[2] and `-fbounds-safety` in Clang[3].

In [P2900R9], a Contracts facility is being proposed for inclusion in C++. One of the explicit goals of this proposal is to make C++ a safer language, and to make C++ programs more safe. The proposal has triggered discussions about whether this goal is being met, some going as far as saying that Contracts are not a safety feature at all: after all, they do not add new language guarantees, and instead focus on correctness checks during program evaluation, with no such checks being performed unless the developer adds explicit contract assertions to their code.

The goal of this paper is twofold: first, to explain how Contracts fit into the picture of making C++ more safe; second, to propose a design direction to make the Contracts facility more effective at detecting and mitigating undefined behaviour in C++, including in code that has no explicit contract annotations at all.

[1] If the variable is default-initialised, of scalar type, and has automatic storage duration.
[2] See https://libcxx.llvm.org/Hardening.html.
[3] See https://clang.llvm.org/docs/BoundsSafety.html.

2 The role of Contracts for Safe C++

There are two possible approaches for removing occurrences of undefined behaviour from C++:

1. Reject the offending constructs at compile time;
2. Give the offending constructs well-defined behaviour during program execution.

In general, approach 1 seems preferable, as it allows the developer to recognise and remove offending constructs earlier in the release cycle and the bug never makes it into the codebase.

However, not all undefined behaviour can be addressed with Approach 1. We can classify occurrences of undefined behaviour into two buckets: one for occurrences for which approach 1 seems feasible, and one for occurrences that can only be addressed by Approach 2 (see Figure 1).

The boundary between these two buckets can be moved towards the right (i.e., by making more undefined behaviour being rejected at compile time) by evolving the C++ Standard towards introducing more constraints on well-formed constructs. This direction, however, is not viable in all cases.

In some cases, the reason why Approach 1 may not be viable is due to the existence of legacy code that needs to keep compiling and working as intended. Rejecting certain constructs could lead to an unacceptable amount of false positives that would break such code; for example, if we were to make it ill-formed to not initialise a variable, or to flow off the end of a non-`void` function. For other forms of undefined behaviour, this approach may call for the introduction of new syntax that makes the offending construct inexpressible, but also requires vast amounts of code to be rewritten — for example, if we were to introduce a Rust-like borrow checker to C++ as proposed in [P3390R0] to address memory and thread safety, which does not permit the usage of regular pointers, references, and random access iterators in safe code.

In other cases, the reason why Approach 1 cannot be applied is because whether the offending construct has undefined behaviour depends on runtime values that are unknowable at compile time. Examples of this are operations on built-in types such as integer overflow, bad bit shifts, division by zero, or unrepresentable arithmetic conversions.

Therefore, we should use Approach 1 whenever feasible and viable; however, there will always be cases where Approach 2 is our only option. Contracts, as proposed in [P2900R9], provide a comprehensive framework for detecting program defects *during evaluation* of a program. Therefore, while Contracts do not help us with Approach 1, they are very well suited for providing a powerful and flexible language tool for Approach 2. As we will see in the remainder of this paper, such a tool

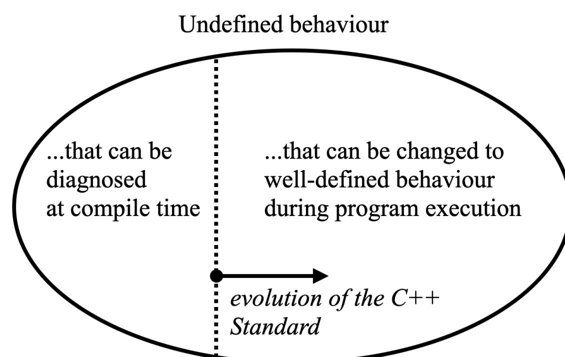

Figure 1: Two different approaches for removing occurrences of undefined behaviour from C++.

emerges naturally if we apply the Contracts framework to *core language constructs* in addition to user-authored contract assertions as proposed in [P2900R9].

Thus, the role of Contracts in improving the safety of C++ is *not* to introduce language constraints to reject unsafe constructs statically, but to *complement* such efforts by providing a mechanism for giving unsafe constructs well-defined behaviour during program execution in cases where the static approach is not technically feasible or does not have favourable tradeoffs for application to existing legacy code.

3 Defining undefined behaviour

In order to understand how Contracts can be applied to detect and mitigate undefined behaviour during program execution, we need to categorise it. In Section 2, we already discussed how some occurrences of undefined behaviour can be removed by evolving the C++ language towards rejecting the offending constructs at compile time, while others cannot. In the remainder of this paper, we focus entirely on the latter category. In this section, we introduce several useful sub-categories; along with it, we develop specification and mitigation strategies within the Contracts framework that are suitable for each such sub-category.

3.1 Library UB vs. Core UB

Sometimes, a distinction is made between *library* undefined behaviour on the one hand, and language or *core* undefined behaviour on the other hand. The C++ Standard itself does not make such a distinction: undefined behaviour is undefined behaviour. In particular, violating the preconditions of a Standard Library function is specified as undefined behaviour.[4] Indeed, a C++ compiler may have special knowledge about its associated Standard Library implementation, and may use that knowledge for certain assumptions and optimisations that would otherwise not be feasible.

However, especially for *non*-standard libraries, the distinction between library and core undefined behaviour is useful. Consider the declaration of a library function with a narrow contract, i.e., a plain-language contract[5] that contains a precondition, such as `operator[]` on some contiguous container:

```
// Returns a reference to the i-th element of the container.
// The behaviour is undefined unless i < size().
T& operator[] (size_t i);
```

If we call this function out of contract (violating its precondition), we cause library undefined behaviour. At this point, the program is defective, and anything can happen; what will happen depends on the implementation of the function. A *narrow implementation* of the function can end up dereferencing a pointer that does not point to an object of the appropriate type within its lifetime, thus leading to *core* undefined behaviour:

```
T& operator[] (size_t i) {
  return _data[i];
}
```

This core undefined behaviour in turn can lead to any behaviour, such as a crash or access into memory unrelated to the container, which in turn creates a security vulnerability. On the other hand, a *wide implementation* of the same narrow-contract function may prevent core undefined behaviour by terminating the program, throwing an exception, etc. before control flow can reach the offending core language construct, for example:

[4] See [structure.specifications]/3.3.

[5] For a definition of the term *plain-language contract* and its relationship with contract assertions, see [P2900R9], Section 2.1.

```
T& operator[] (size_t i) {
  if (i >= size())
    __builtin_trap(); // fail hard and fast! – or, throw an exception, trigger a breakpoint, etc.

  return _data[i];
}
```

The tradeoff is thus between eliminating the possibility of core undefined behaviour at the cost of an extra runtime check. Whether that tradeoff is the correct one will depend on the context — there can be no universal answer for every C++ program. It is therefore necessary to offer different options to users.

Importantly, the implementer of a library containing the above function, the developer of the application calling the function, and the person compiling and linking the final program may be completely independent parties. Therefore, choice between narrow and wide implementations should be made at the stage of compiling and linking the final program, *not* in the code itself. This is existing practice: C `assert` offers this choice via the `NDEBUG` flag; libc++ offers this choice via different hardening modes; etc.

[P2900R9] improves on the above approach in a number of ways. First, it allows for the precondition to be expressed on the declaration of the function, rather than inside its definition; second, it standardises the available options by offering a set of four *evaluation semantics* that can be chosen between in an implementation-defined way: *ignore*, *observe*, *enforce*, and *quick_enforce*. With [P2900R9] contract assertions, the declaration of the function above becomes:

```
T& operator[] (size_t i)
pre (i <= size());
```

[P2900R9] thus makes C++ safer by offering the developer a powerful tool to detect and mitigate library undefined behaviour *before* it can progress to cause core undefined behaviour — and thus unbounded undefined behaviour that can cause safety issues and security vulnerabilities — without sacrificing flexibility for use cases where such detection and mitigation is not affordable or otherwise desired.

However, in order to get this benefit from [P2900R9] as proposed, the user actually needs to add the contract assertion `pre (i <= size())` to their code, or use a library that does this. Contracts as proposed in [P2900R9] thus directly address library undefined behaviour, but only indirectly address core undefined behaviour.

The central idea of our proposal is to extend C++ in two ways:

— Introduce the notion of *implicit contract assertions* that are inserted by the implementation rather than provided by the user, but otherwise have the same semantics as the *explicit* contract assertions proposed in [P2900R9];

— In the core language, change every occurrence of “Operation X has undefined behaviour if A is not `true`” to “Operation X has an implicit precondition that A is `true`”.

As an example, let us consider an out-of-bounds access on a *raw* array rather than a library class. Let us assume for the purpose of this example that the size `N` of this array is statically known:

```
int main() {
  int a[10] = { /* ... */ };
  std::size_t i; std::cin >> i;
  return a[i];
}
```

As written, this program has potential undefined behaviour if the value of `i`, which is not statically known, is not smaller than `10`. On the other hand, with our proposal, the program behaves as-if

the compiler had wrapped every raw array subscript operation for which it statically knows the array bound `N` into an inline function with a precondition assertion:

```
template <typename T, std::size_t N>
T& __index_into_array(T (&a)[N], std::size_t i)
pre (i < N) {
  return a[i];
}
```

Other than being an *implicit* precondition assertion generated by the compiler, `pre (i < N)` behaves as specified in [P2900R9]. That is, the user has the choice whether or not to check this assertion and whether the contract-violation handler (which is the same as for explicit contract violations) should be called; this choice happens via implementation-defined means, e.g., a compiler flag.

Thus, the Contracts framework can be directly applied to core language undefined behaviour, and therefore to legacy code, without the need to add any explicit contract assertions. The implementation can choose which evaluation semantics it provides for each kind of implicit contract assertion; if it is not capable of generating checks for certain kinds of core language operation with implicit contract assertions, it can choose to not offer a checked contract semantic for this kind of contract assertion while still being conforming.

3.2 UB vs. EB

[P2795R5] introduced to C++ the concept of *erroneous behaviour*, that is, behaviour that is well-defined yet known to only occur when there is a defect in the program. It therefore allows the C++ Standard to talk about *incorrect* programs, i.e., programs that behave counter to the intent of the author when executed.

Note that the concept of a *contract violation* in [P2900R9] does exactly the same thing. The salient difference, until now, was that erroneous behaviour applies to misbehaving core language constructs, whereas contract violations occur when explicit contract assertions are checked and determined to not evaluate to `true`. However, with our proposed extension, contract violations can also occur when core language constructs violate their *implicit* contract.

This leads us to two key insights. The first is that erroneous behaviour is simply another term for implicit contract violation — they are, conceptually, *one and the same thing*. The second is that the semantics of erroneous behaviour as specified in [P2795R5] are a strict subset of the possible evaluation semantics of a contract violation. According to that specification, when erroneous behaviour occurs, the implementation is permitted to do any of the following:

— Issue a diagnostic;

— Terminate the application;[6]

— Do nothing.

Note that all the above options are covered by the four evaluation semantics in [P2900R9], *ignore*, *observe*, *enforce*, and *quick_enforce*. In addition to the above options, these evaluation semantics add more possibilities, in particular calling the contract-violation handler, which is a powerful tool to manage mitigation strategies and avoid unwanted unconditional program termination.

[6]There is a subtle difference in how program termination is defined for erroneous behaviour on the one hand, and for terminating semantics in [P2900R9] on the other hand. The wording adopted from [P2795R5] specifies that on erroneous behaviour, the implementation is permitted to terminate the execution at an unspecified time after that operation, whereas the terminating semantics in [P2900R9] do not give permission for the termination to be delayed until an unspecified time. One of the reasons why it is desirable to unify both frameworks, rather than having two separate mechanisms for achieving essentially the same thing, is that we can avoid such subtle divergences in behaviour.

Having two separate language mechanisms that are trying to achieve the same thing does not seem to be a sound design. We therefore propose to remove the notion of *erroneous behaviour* from C++ entirely and replace it with the notion of implicit contract assertions as proposed here.

Concretely, we propose to change every occurrence of "Operation X has erroneous behaviour if A is not `true`" to "Operation X has an implicit precondition that A is `true`." This works not only for reading erroneous values, which is already specified as erroneous by [P2795R5], but also for every other occurrence of undefined behaviour in C++ listed as "could be changed to erroneous" in [P2795R5].

With this change, the function `std::erroneous()` as proposed in [P3232R0] becomes simply a shorthand for `contract_assert(false)`, much in the same way as `std::unreachable()` today is a shorthand for `[[assume(false)]]`.

Note that we do not propose to remove the notion of *erroneous value*; however, instead of specifying that reading an erroneous value is erroneous behaviour, we propose to specify that reading a value has an implicit precondition that the value is not erroneous.

3.3 UB with vs. without safe fallback behaviour

Undefined and erroneous behaviour in C++ today can be classified according to whether or not a *safe fallback behaviour* can be reasonably defined for a particular occurrence of such behaviour. This distinction is one of the two dimensions shown in Figure 2 (we will discuss the other dimension further below). This property is important, as it determines which evaluation semantics can be applied to which kinds of implicit contract assertions.

Occurrences of undefined behaviour for which we *can* define safe fallback behaviour include integer overflow, bad bit shifts, division by zero, and unrepresentable arithmetic conversions. For example, for signed integer overflow, a safe fallback behaviour is that the result of the addition will be some valid number. We could additionally specify the value of this number, for example, by specifying that signed integer overflow wraps or saturates, but to remove the undefined behaviour it is entirely sufficient to say that it will be an unspecified but valid value. Using this number in a calculation will most likely be incorrect (i.e. a bug), but it will no longer be undefined.

	easy to check during program execution	*hard to check during program execution*
safe fallback behaviour does not exist	• Out-of-bounds access (bound known) • Flowing off the end of a non-void function • Calling pure virtual function in an abstract base class constructor or destructor	• Out-of-bounds access (bound unknown) • Accessing object outside its lifetime • Accessing object through pointer or reference of wrong type (aliasing violations) • Modifying a const value
safe fallback behaviour exists	• Integer overflow • Bad bit shifts • Division by zero • Unrepresentable arithmetic conversions • Reading an erroneous value	• Data race

Figure 2: Categories of undefined behaviour that can be respecified in terms of implicit contract violations

We can thus specify that each signed integer addition in the program behaves as-if the compiler had wrapped it into an inline function with the following precondition assertions:

```
int operator+(int a, int b)
pre ((b >= 0 && a <= INT_MAX - b) || (b < 0 && a >= INT_MIN - b));
```

and that the value returned by this notional function is the sum of the two integers, or, if the operation overflows, some unspecified but valid integer. Thus, we redefine the behaviour of an operation that today has undefined behaviour, but has a reasonable safe fallback behaviour, to be *that safe fallback behaviour*, guarded by the appropriate implicit contract assertions. We have thus simultaneously removed the undefined behaviour and introduced the necessary correctness checks to detect bugs due to overflow.

Note that all existing strategies for handling signed integer overflow in C++ today map exactly to one of the possible five contract evaluation semantics (the four proposed in [P2900R9] plus the *assume* semantic; see Section 4.1). Such strategies include:

- The GCC compiler option `-ftrapv`, which aborts the program on signed integer overflow, is a conforming implementation of the *quick_enforce* semantic;
- A sanitiser which detects signed integer overflow and prints a diagnostic is a conforming implementation of the *enforce* or *observe* semantic (depending on whether the process is terminated or execution continues after printing the diagnostic);
- The GCC compiler option `-fwrapv`, which implements signed integer addition using wrap around using twos-complement representation, is a conforming implementation of the *ignore* semantic, which silently executes the safe fallback behaviour;
- The default behaviour in C++ today, which is to assume that signed integer addition can never overflow, and optimise based on this assumption when the appropriate optimisation flags are selected by the user, is a conforming implementation of the *assume* semantic.

Now, let us consider an occurrence of undefined behaviour for which *no* safe fallback behaviour can be reasonably defined. One example is out-of-bounds access into a raw array. What should such access return? It could return a default-constructed value of the element type, but the type may be not default-constructible. It could return an erroneous value, similar to an uninitialised variable, but that would not work for non-scalar types.

The answer is that it should not return at all; there should be no circumstances in which control flow continues past a violation of such an implicit contract assertion. Given that it might not always be feasible to actually detect such a violation (e.g., if the bound of the array is not statically known), this leaves three out of five contract evaluation semantics as viable options:

- Implementations that diagnose violations of the implicit contract assertion during program execution (e.g., AddressSanitizer) can implement the *enforce* or *quick_enforce* semantics;
- The default behaviour in C++ today, which is to assume that a violation can never happen, and optimise based on that assumption, is a conforming implementation of the *assume* semantic.

We propose to define such operations that have no reasonable defined behaviour in case of a violation of their implicit contract as having *structurally narrow* implicit preconditions. A structurally narrow implicit precondition assertion specifies a precondition on a narrow-contract operation that cannot have a wide implementation. An implicit contract assertion that does not have this property is a *structurally wide* implicit contract assertion.

A structurally narrow implicit precondition cannot be evaluated with the *ignore* or *observe* semantic, as there is no possible behaviour for the program for these semantics; the only possible options are

to enforce or to assume that such behaviour can never occur. On the other hand, structurally wide implicit contract assertions can be evaluated with any evaluation semantic.

3.4 UB that is easy vs. difficult to diagnose during program execution

The other axis in Figure 2 that is useful for classifying occurrences of undefined behaviour is how *easy* or *difficult* it would be to check for violations of the implicit contract of the affected operation. By *easy* we mean an implicit contract assertion for which we can expect that a regular C++ compiler will be able to lay down runtime checks by choosing to do so with any granularity desired, without any need for cross-program instrumentation or similarly herculean implementation efforts.

Examples of structurally narrow implicit contract assertions that are *easy* to check include out-of-bounds access into a raw array if the bound of the array is known at compile time; flowing off the end of a non-`void` function; and calling a pure virtual function in the constructor or destructor of an abstract base class. Examples of structurally wide implicit contract assertions with this property are integer overflow, bad bit shifts, division by zero, and unrepresentable arithmetic conversions, as well as reading an erroneous value.

Note that easy does not necessarily mean computationally cheap. A bounds check on array access can be insignificant in many cases; on the other hand, mitigating all occurrences of signed integer overflow can have an unacceptable performance impact for numerically heavy applications. However, in the framework of [P2900R9], the semantic of each evaluation of a contract assertion is implementation-defined and in principle independent of the semantic of any other evaluation. A conforming implementation can therefore offer separate compiler flags to turn on bounds checks and to turn on defined behaviour for signed integer overflows, respectively. This is by and large already existing practice; our proposal merely formalises this existing practice by phrasing it in terms of contract evaluation semantics, and adds extra options such as calling the contract-violation handler (see Section 5.2).

Examples of implicit contract assertions that are *difficult* to check are most memory safety issues such as accessing an object outside its lifetime, accessing an object through a pointer or reference of the wrong type, and out-of-bounds access into a raw array if the bound of the array is *not* known at compile time; as well as data races. By *difficult* we mean that a regular C++ compiler probably will *not* have the means to lay down the appropriate runtime checks, as doing so would require extensive instrumentation. Offering *checked* semantics for such implicit contract assertions would therefore most typically only be offered by specialised implementations of C++ such as sanitisers (AddressSanitizer, ThreadSanitizer, etc.); our proposal would allow to reason about the semantics of such tools within the C++ Standard, and hook them into the program-wide contract-violation handler (see Section 5.3).

3.5 Explicit vs. implicit UB

In addition to *explicit* undefined behaviour, i.e. operations for which the C++ Standard explicitly states that the behaviour is undefined, there is also *implicit* undefined behaviour, which occurs when the C++ Standard does not say anything at all about the runtime semantics of a particular operation.

This proposal is focused on explicit undefined behaviour, because we can specify concrete implicit contract assertions for such operations. However, we do not prevent an implementation from treating implicit undefined behaviour as a contract violation as well, if it is capable of detecting such undefined behaviour during program evaluation.

4 Adding the missing pieces

4.1 The *assume* semantic

We already mentioned the fifth contract evaluation semantic, *assume*, in previous sections. While this semantic is not included[7] in the set of semantics proposed in [P2900R9], it is an important piece of the puzzle for two reasons.

The first reason is that there are occurrences of undefined behaviour — or, with our proposal, implicit contract assertions — that we cannot expect every implementation of C++ to be able to check at runtime. Some of these occurrences will be operations for which there is no safe fallback behaviour — in other words, there is no well-defined behaviour that these operations could possibly have if control flow continues past violations of their implicit preconditions. For such operations, if the compiler cannot lay down checks that can be used to *enforce* the preconditions, the only other option is to compile the program with the *assumption* that the check will never fail, so no instructions for this case need to be emitted by the compiler. This is the status quo today for undefined behaviour in the top right quadrant of Figure 2, for example undefined behaviour due to accessing an object outside its lifetime.

The second reason is that even for implicit contract assertions that do have a safe fallback behaviour — with our proposal, this is the behaviour that will be executed when the semantic is *ignore* — such fallback behaviour might lead to measurable performance degradations. This creates a problem for applications that care first and foremost about performance, and less about safety and security, such as high-performance numerical simulations and low-latency signal processing. Such applications must have an escape hatch that allows them to avoid degradations and preserve today's behaviour, in which the affected implicit contract assertions are *assumed* rather than *ignored*, with the consequence that contract violations may lead to core undefined behaviour.

For reading an indeterminate value, [P2795R5] offers such an escape hatch in the form of the new attribute `[[indeterminate]]`, which disables the safety guardrail and restores the C++23 behaviour. However, offering such an explicit, syntactic escape hatch for every possible case of undefined behaviour does not scale; it would make the language unnecessarily complex, and in some cases, would even not be possible — how would we annotate, for example, an expression containing signed integer addition, and would we want to do that throughout our entire numerics library?

Our approach is therefore to offer a generic escape hatch in the form of the *assume* semantic. This semantic can be selected by the user for a particular kind of implicit contract assertions, for example to squeeze maximum performance out of their numeric operations, while the other four semantics (none of which leave any room for undefined behaviour) can be used for other kinds of implicit contract assertions, for example to enable checks that guard against flowing off the end of non-`void`-returning functions. This framework enables the expert user to select precisely which tradeoffs between safety and performance are most appropriate for their use case; for the average user, implementations can offer reasonable defaults.

4.2 Library API extension

Re-specifying undefined and erroneous behaviour as a contract violation as proposed here means that, if any of the introduced implicit contract assertions is determined to be violated during

[7] One of the reasons why SG21 decided to not include the *assume* semantic in the first version of the proposed Contracts facility is that it has the potential to introduce undefined behaviour to an otherwise correct program via a buggy contract predicate. The presence of the *assume* semantic in the C++2a Contracts proposal [P0542R5] was controversial at the time and contributed to that proposal being removed from the C++20 Working Draft. An assumption facility was nevertheless seen as an important feature, and was subsequently added to C++ separately from the Contracts facility in the form of the `[[assume]]` attribute (see [P1774R8]).

program execution, and the selected evaluation semantic is *observe* or *enforce*, the contract-violation handler will be called. In order to accommodate implicit contract assertions in the associated library API for contract-violation handling, we need to make some minor additions to the contract-violation handling API introduced by [P2900R9].

We propose to add a new enum value `implicit` to the enum `assertion_kind`. This value will be returned from the `kind()` member function of the `contract_violation` object passed into the handler when the contract violation that caused the call to the handler resulted from an implicit rather than an explicit contract assertion. Note that we do not attach any particular meaning to the numerical value of `implicit`; in particular, we do not define it to be `0` because we believe it is useful to be able to detect the case where the enum has not been explicitly initialised with a valid value.[8]

We further propose that the value returned by `detection_mode()` for violations of implicit contract assertions be unspecified. This choice gives implementations freedom to specify detection modes that are best suited to express the mitigation strategies they employ, and to inform the user of the strategy used to detect a particular implicit contract violation.

We do not propose any changes to the specification of `comment()` and `location()`. [P2900R9] non-normatively recommends that these functions return a textual representation of the expression that triggered the contract violation and the source location of the contract violation, respectively. The same is in principle possible for violations of implicit contract assertions. However, generating a textual representation for every possible expression that could lead to diagnosable undefined behaviour is likely to cause an unacceptable amount of code bloat. It is therefore equally conforming to instead generate some other string that may help the user identify the problem, such as the diagnostic message already printed by existing sanitisers. It is further conforming to simply return an empty string and a default-constructed source location if no information is available, or if the information is not programmatically accessible in the contract-violation handler (for example, because it located in a separate debug information file).

5 Discussion

5.1 Contracts provide a framework for safety

One benefit of adopting the framework of implicit contract assertions as proposed here for C++ is that it provides users, vendors of compilers and tools, as well as the C++ Standard committee with the vocabulary needed to reason about and to specify different options for the detection and mitigation of undefined behaviour.

In particular, once we re-specify undefined behaviour in terms of implicit contract assertions, the [P2900R9] framework specifies that any evaluation of such an assertion can have any evaluation semantic, and that the choice of semantic is implementation-defined, meaning that implementations are expected to document which semantics they support for which implicit contract assertions, and which selection mechanism they offer.

For example, a compiler that adds checks for out-of-bounds access no longer has to describe such checks as a vendor extension; it can simply document that it offers checked contract semantics for accessing an array. A user who wishes to enable such checks can more easily find tools that offer them, and reason about the semantics that these tools provide.

Further, coding guidelines can place restrictions on which semantics are permitted for which kinds of implicit contract assertions. For example, in a safety-critical context, a set of coding guidelines may

[8] See also [P3227R0], which proposes to additional enum values to the enum `evaluation_semantic` and makes the same argument.

prescribe that *assume* semantics may not be used for certain kinds of implicit contract assertions, thus preventing the usage of toolchains and compiler options that could lead to the program exhibiting a particular kind of undefined behaviour — provided of course that alternatives which offer semantics other than *assume* for those implicit contract assertions actually exist.

Finally, re-specifying undefined behaviour in terms of implicit contract assertions would be a visible step by the C++ committee towards more safety in C++. While such re-specification in itself does not make any code safer unless compilers and tools actually offer ways to perform the associated checks, it would make such efforts more visible and explicit because they would be happening within a unified framework specified by the C++ Standard.

5.2 Contracts provide a program-wide handler

Another benefit of the approach proposed here is that any compiler and tool that adds checks to detect and mitigate undefined behaviour — or, in the language of our proposal, violations of implicit contract assertions — during program execution could hook into the program-wide contract-violation handler defined by [P2900R9]. This would allow the user to treat all possible defects that are detected during program execution in one place.

As explained in [P2698R0], in many situations unconditional program termination is simply not a viable strategy for handling bugs. Providing a single hook for users to change their mitigation strategy to something else makes the entire language more coherent and supportable. Even in situations where termination is the desired goal, reporting an bug that has been encountered in a consistent manner greatly increases the chances of fixing the bug promptly and hardening software against further unsafe operations or undesired terminations in the future.

Simply terminating with the *quick_enforce* semantic, or with a bespoke error message or distinct tool-specific handling mechanism, becomes quickly unmanageable as more and more different tools and detection mechanisms are employed. By standardising on a central reporting mechanism we separate clearly the responsibility for reporting from the responsibility of knowing all of the mechanisms within a program by which a bug might be detected.

We therefore do not introduce a separate contract-violation handler for implicit contract assertions; if the user really wishes to employ a different reporting or mitigation strategy for defects due to violations of implicit contract assertions than for those due to violations of explicit contract assertions, they can always branch on the `assertion_kind` (see Section 4.2) and dispatch to custom handlers for these cases.

5.3 Contracts provide better integration for sanitisers

The ability to interface with the Contracts machinery and the contract-violation handler applies not only to C++ compilers, but also extends to sanitisers and other tools, which might be able to offer checked semantics for a much wider range of implicit contract assertions.

The integration between such tools and user code today is poor. For example, all clang sanitisers have a callback `__sanitizer_set_death_callback`, but this callback takes no arguments. It can be used to inform the user that the process is about to terminate, but it does not provide an API to programmatically query what happened or where. ASan has a slightly more sophisticated callback `__asan_set_error_report_callback` which takes a single argument of type `const char*`. This argument provides a string that contains the generated error report.

With our proposal, all these tools can instead hook into the API provided in [P2900R9], taking advantage of the ability to provide new values for the various enumerations that describe a contract violation, as proposed in Section 4.2. This API provides not only a user callback in the form of a program-wide replaceable contract-violation handler, but also programmatically accessible

information about the defect via the `contract_violation` object passed into the contract-violation handler. This more comprehensive API can serve as a uniform standard callback mechanism for sanitisers and other tools.

Note that supporting this API is entirely optional for existing sanitisers. For those occurrences of undefined behaviour — or, in the language of our proposal, violations of implicit contract assertions — that the sanitiser can diagnose, it can implement the *enforce* semantic to call into the contract-violation handling API as described above, or a bespoke enforce-like semantic which calls their present-day death hook; for those that it cannot diagnose, it can implement the *ignore* semantic, or, for structurally narrow implicit contract assertions, the *assume* semantic. In all cases, the sanitiser will be standard-conforming with this proposal; the only additional requirement is that the choice of semantic for each kind of implicit contract assertion will need to be documented. Note that this choice of semantic can vary between different implicit contract assertions of the same kind, or even different evaluations of the same such implicit contract assertion: for example, ThreadSanitizer can report certain occurrences of data races but not others.

6 Proposal summary

Below we provide an informal summary of the proposed changes, which are on top of those proposed in [P2900R9]. Formal wording can be added after the design direction in this paper has been confirmed by the relevant study groups.

- Add the notion of *explicit* contract assertion for a contract assertion specified by the user, which includes all three kinds of contract assertions in [P2900R9] — `pre`, `post`, and `contract_assert`.
- Add a fifth contract evaluation semantic, the *assume* semantic, to the four semantics in [P2900R9]. The *assume* semantic is an unchecked semantic, i.e. it does not attempt to determine the value of the predicate. If the predicate would not evaluate to `true` at the point of evaluation of the contract assertion, the behaviour is undefined.
- Add the notion of an *implicit* contract assertion, that is, a contract assertion implicitly generated around a certain language construct by the implementation rather than specified by the user with `pre`, `post`, or `contract_assert`.
- Add a note that implicit contract assertions can be evaluated with any of the five contract evaluation semantics, just like explicit contract assertions.
- Remove the notion of *erroneous behaviour*, replace it with "violation of an implicit contract assertion", and transform affected specifications as follows:
  - Replace the specification that reading an erroneous value is erroneous behaviour with the specification that reading a value has an implicit precondition that the value is not erroneous;
  - ...
- Transform occurrences of undefined behaviour for which a safe fallback behaviour can be specified to instead be phrased in terms of implicit contract assertions, for example:
  - Change signed integer overflow/underflow from being undefined behaviour to producing an unspecified value; add to the specification of the associated signed integer operations that they have an implicit precondition that no overflow/underflow will occur;

- Change shifting by equal or greater than the bit-width of a type from being undefined behaviour to producing an unspecified value; add to the specification of shift operations that they have an implicit precondition that the right operand is smaller than the bit-width of the left operand;
  - Change arithmetic conversions where the source value cannot be represented in the destination type from being undefined behaviour to producing an unspecified value; add to the specification of arithmetic conversions that they have an implicit precondition that the source value can be represented in the destination type;
  - ...

- Add the notion of *structurally wide* and *structurally narrow* implicit contract assertions. A structurally narrow contract assertion cannot be evaluated with the *ignore* or *observe* semantics, as there is no well-defined behaviour possible if control flow were to continue past a violation of a structurally narrow contract assertion.

- Transform occurrences of undefined behaviour for which no safe fallback behaviour can be specified to instead be phrased in terms of structurally narrow precondition assertions, for example:

  - Replace the specification that accessing an object outside of its lifetime is undefined behaviour with the specification that accessing an object has a structurally narrow implicit precondition that the object is within its lifetime;
  - Replace the specification that attempting to access the stored value of an object through a glvalue through which it is not type-accessible is undefined behaviour with the specification that accessing the stored value of an object through a glvalue has a structurally wide implicit precondition that the object is type-accessible through that glvalue;
  - Replace the specification that attempting to modify a `const` object is undefined behaviour with the specification that any modification of an object has a structurally narrow implicit precondition that the object is not `const`;
  - ...

- Extend `std::contracts::assertion_kind` as follows:

  ```
  enum class assertion_kind {
   pre = 1,
   post = 2,
   assert = 3,
   implicit = 4
  };
  ```

  where the added enum value stands for implicit contract assertions.

- Add a provision that `std::contracts::contract_violation::detection_mode()` returns an implementation-defined value if the violated contract assertion was an implicit contract assertion.

Bibliography

[Abrahams2023] David Abrahams. Values: Safety, Regularity, Independence, and the Future of Programming (CppCon 2023). https://www.youtube.com/watch?v=QthAU-t3PQ4, 2023-01-07.

[Bastien2023] JF Bastien. Safety and Security: The Future of C++ (C++Now 2023 Keynote). https://www.youtube.com/watch?v=Gh79wcGJdTg, 2023-07-14.

[CISA2023] Cybersecurity and Infrastructure Security Agency. The Case for Memory Safe Roadmaps. https://www.cisa.gov/sites/default/files/2023-12/T, 2023-12.

[CR2023] Consumer Reports. Report: Future of Memory Safety. https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report-1-1.pdf, 2023-01-22.

[Carruth2023] Chandler Carruth. Carbon Language Successor Strategy: From C++ Interop to Memory Safety (C++Now 2023 Keynote). https://www.youtube.com/watch?v=1ZTJ9omXOQ0, 2023-09-08).

[Doumler2023] Timur Doumler. C++ and Safety (C++North 2023). https://www.youtube.com/watch?v=iCP2SFsBvaU, 2023-09-22.

[NSA2022] National Security Agency. Cybersecurity Information Sheet – Software Memory Safety. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF, 2022-11.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. https://wg21.link/p0542r5, 2018-06-08.

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

[P1774R8] Timur Doumler. Portable assumptions. https://wg21.link/p1774r8, 2022-06-14.

[P2012R2] . . https://wg21.link/p2012r2, 20.

[P2698R0] Bjarne Stroustrup. Unconditional termination is a serious problem. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2698r0.pdf, 2022-11-18.

[P2795R5] Thomas Köppe. Erroneous behaviour for uninitialized reads. https://wg21.link/p2795r5, 2024-03-22.

[P2809R3] Nicolai Josuttis, Victor Zverovich, Filipe Mulonde, and Arthur O'Dwyer. Fix the range-based for loop, Rev 2. https://wg21.link/p2809r3, 2020-09-29.

[P2900R9] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r9, 2024-10-11.

[P3227R0] Gašper Ažman and Timur Doumler. Fixing the library API for contract violation handling. https://wg21.link/p3227r0, 2024-10-15.

[P3232R0] Thomas Köppe. User-defined erroneous behaviour. https://wg21.link/p3232r0, 2024-04-16.

[P3274R0] Bjarne Stroustrup. A framework for Profiles development. https://wg21.link/p3274r0, 2024-05-05.

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