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 #:	P3100R0
Date:	2024-05-21
Project:	Programming Language C++
Audience:	EWG, SG21, SG23

Abstract

In this paper, we propose to specify undefined behaviour that manifests at runtime to be a *contract violation*. Core language constructs that can exhibit undefined behaviour are respecified as having *implicit* contract assertions that can be evaluated with the same evaluation semantics as the *explicit* contract assertions proposed in the Contracts MVP ([P2900R7]). We further propose to add the `assume` semantic to the four evaluation semantics in [P2900R7]. Violating an assertion with the `assume` semantic is undefined behaviour and is, for operations with implicit contract assertions, equivalent to the behaviour of those operations in C++23, thus providing backwards-compatibility without performance regressions. The other four evaluation semantics map to different strategies to reduce the probability of undefined behaviour at runtime, such as hardened builds and sanitisers. Our proposed framework replaces the concept of *erroneous behaviour* introduced in [P2795R5] and paves the way for a safer and more secure C++ Standard.

1 Introduction

1.1 Safety and security in C++

Improving safety and security is one of the main challenges for the future evolution of the C++ language ([Bastien2023]). 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.

According to these definitions, C++ is not a safe language[1]: both the core language and the C++ standard library allow for a multitude of ways in which a well-formed C++ can exhibit unbounded undefined behaviour, that is, situations in which the C++ standard places no restrictions on the behaviour of the program. Bugs that trigger undefined behaviour cause stability issues and can be exploited by threat actors, thereby causing security vulnerabilities.

[1] Note that our usage of the terms "safe language" and "safe operation" meaning "no unbounded undefined behaviour in the face of bugs" contrasts with the concept of *functional safety*, which is closer to the colloquial usage of the term "safe" and can be defined as the property of a system — rather than a computing operation or a programming language — to not cause real harm in the face of bugs. Under the latter definition, there is no "safe language", only some languages that are easier to use safely than others.

Of particular concern is undefined behaviour due to invalid memory accesses, which was found by large companies such as Google and Microsoft to be responsible for ~70% of security vulnerabilities ([NSA2022], [CR2023]). This led various government agencies to start advocating for an industry-wide shift away from C and C++ and towards memory-safe languages such as Rust ([CISA2023], [ONCD2024]), triggering a debate about the future of C++.

1.2 Current approaches

Various approaches exist to reduce the probability of undefined behaviour occurring in a C++ program during runtime. Some of them are good engineering practices that are not specific to C++, such as code reviews, enforcing coding guidelines such as MISRA or the C++ Core Guidelines, and automated testing targeting different aspects of the code (unit tests, fuzz tests, etc). When applied correctly, these techniques can significantly reduce the amount of undefined behaviour that can occur during runtime. In addition, it can be effective to use static analysis tools (such as Coverity, Sonar, PVS-Studio, etc.) in order to statically detect reachable code-paths which exhibit undefined behaviour if executed, or at runtime, by using a sanitiser (such as ASan and UBSan) to stop the program in case it detects such a violation of the rules of the abstract machine.

In addition, the ISO C++ committee, as the body owning the specification of the C++ programming language, is in the unique position to be able to evolve this specification to eliminate sources of undefined behaviour *normatively* for all conforming implementations of C++. Given that undefined behaviour in C++ is fundamentally a runtime property, two approaches are possible. First, by making the program ill-formed if we can statically detect a code path that would exhibit undefined behaviour if taken. Second, by specifying the range of behaviours that such a code path can have once it breaches the defined behaviour of the abstract machine, therefore mitigating it at runtime. These two approaches complement each other.

1.3 Runtime handling of undefined behaviour

In principle, it would be possible to evolve C++ such that by default, undefined behaviour can no longer occur, because any potentially unsafe construct would be rejected at compile time. The Rust programming language is a successful proof of concept for this approach. However, doing so would break practically every useful program that exists in C++ today, and therefore does not seem to be a realistic prospect [Doumler2023].

For example, in order to reject all invalid memory accesses at compile time, we would need to make any code ill-formed that allows multiple pointers or references to the same object to exist, unless it can be statically proven that all such pointers or references are `const`. This effectively requires an approach like Rust's borrow checker or Hylo's mutable value semantics, thereby disqualifying any C++ code that returns a non-`const` reference from a function, uses random access, doubly-linked lists, etc. This might be viable for some *new* code (where potentially unsafe constructs could be contained in *unsafe blocks* and kept to a minimum like in Rust, and the rest of the code written using the safe approach), but not for *existing* code.

Therefore, to address safety in C++, we should follow a two-pronged approach. On the one hand, we should aim to make more undefined behaviour diagnosable at compile time, wherever the tradeoffs for doing so are favourable. On the other hand, we also need to accept that some undefined behaviour will remain undiagnosable at compile time for the foreseeable future, and will have to be handled at runtime instead. The aim of this paper is to focus entirely on such runtime handling, while leaving compile-time detection to other ongoing efforts such as the safety profiles proposed in [P2687R0].

1.4 The role of Contracts

[P2900R7] proposes to add a Contracts facility to C++. While this proposal is currently only a minimum viable product (MVP) with a limited number of features, it has been explicitly designed with extensibility in mind. This paper proposes one such extension.

It has been stated that such an extensible Contracts facility is a panacea that has the potential to solve the undefined behaviour problem in C++. This is a misconception. Contracts are the wrong tool for making all undefined behaviour diagnosable at compile time. However, for all the undefined behaviour *not* diagnosed at compile time, Contracts provides a comprehensive framework for handling such undefined behaviour at runtime. The key idea is to respecify specific kinds of undefined behaviour to be a *contract violation*. This allows us to formally speak of incorrect code and to define its behaviour in the C++ Standard, rather than leaving the behaviour undefined. This behaviour can be configured by the user (issue a diagnostic message, terminate the program, ignore the contract violation, etc.) by choosing one of several standard *evaluation semantics*.

Recently, the committee has adopted the proposal ([P2795R5]) into the C++26 working draft, which follows essentially the same motivation. The proposal makes reading the value of an uninitialised variable well-defined by respecifying such reads as *erroneous behaviour*, another term for behaviour that is incorrect, but not undefined. When encountering erroneous behaviour, an implementation is allowed and encouraged to diagnose the defect, but is also allowed to ignore it and treat the read as valid — a variation of having different contract evaluation semantics. The paper goes on to say that other undefined behaviour can be respecified to be erroneous behaviour instead.

As we will show in this paper, the semantics provided by erroneous behaviour are essentially a subset of the semantics that the Contracts framework provides. We can therefore entirely subsume erroneous behaviour into contract violation handling. As a result, we get a unified framework for reasoning about and handling runtime undefined behaviour that is more generic and flexible than erroneous behaviour and has several other advantages over the approach proposed in [P2795R5] that is currently in the C++26 Working Draft. Our proposed framework also addresses the last use case wg21.otherfeatures listed in [P1995R1], the "Contracts — Use Cases" paper that laid the foundation for SG21's work.

2 The Contracts MVP

2.1 Explicit contract assertions

The current Contracts MVP [P2900R7] proposes to add *contract assertions*, i.e. syntactic constructs such as `pre(x)`, `post(x)`, and `contract_assert(x)`, where `x` is a boolean expression, to express conditions that are expected to be true in a correct program. For example, we can annotate the subscript operator in a `vector` class with a precondition assertion to catch out-of-bounds access, one of the most commonly exploited sources of security vulnerabilities due to runtime undefined behaviour:

```
T& vector::operator[](size_t index)
pre (index < size()) {  // explicit contract assertion
  return data[index];
}
```

In the remainder of this paper, we will call the contract assertions proposed by [P2900R7] *explicit contract assertions* as they are code explicitly added by the user.

2.2 Evaluation semantics

Contract assertions can be either ignored or checked. When checked, the predicate `x` is evaluated; if it evaluates to `false`, a contract violation has been detected. In general, a contract violation means

that the program is *incorrect* (i.e. has a bug), but the behaviour of the program is nevertheless well-defined at this point.

What exactly happens when a contract assertion is evaluated is determined by the chosen *contract evaluation semantic*. [P2900R7] proposes four such semantics. These are, from strictest to least strict:

— `quick_enforce`: evaluate the contract predicate; if a contract violation occurs, immediately terminate the program.

— `enforce`: evaluate the contract predicate; if a contract violation occurs, call the contract-violation handler. When the contract-violation handler returns, terminate the program.

— `observe`: evaluate the contract predicate; if a contract violation occurs, call the contract-violation handler. When the contract-violation handler returns, continue execution.

— `ignore`: do not evaluate the contract predicate.

We call `ignore` a *non-checking* semantic and the other three *checking* semantics. Additionally, we say that `quick_enforce` and `enforce` are *enforcing* semantics, i.e. semantics that will not allow the code to continue execution past a contract violation. This prevents the program from continuing into code that might have undefined behaviour. Consider, for example, a program that causes an out-of-bounds access with our implementation of `vector::operator[]` from above:

```
int main() {
  vector<int> v;
  std::cout << v[666];     // no UB here if using an enforcing semantic!
}
```

Note that the above is different from using a function without preconditions such as `vector::at` because C++ does not force us to use an enforcing semantic; if we choose a non-enforcing semantic such as `ignore` or `observe`, the behaviour of the above program is undefined. Whether the implementation of `vector::operator[]` is *safe* therefore depends on the chosen evaluation semantic. In a safety-critical context, we could require that an enforcing semantic must be chosen to compile the program. However, using contract assertions in this way does not preclude compiling the code with different semantics in different contexts, and does not require macros to achieve such flexibility.

3 The proposal

3.1 Implicit contract assertions

The core idea of this proposal is that we can apply the Contracts framework not only to explicit contract assertions, but also to core language constructs that may have undefined behaviour when evaluated, by treating such undefined behaviour as a contract violation.

Consider, for example, signed integer addition. When adding two `int`s `a` and `b`, instead of saying that the behaviour is undefined if the result of the addition is not representable by `int`, we can say that signed integer addition has an *implicit precondition assertion* that the addition will not overflow. In other words, if we consider signed integer addition to be performed by notionally calling a built-in `operator+(int, int)`, this built-in operator will behave as if it was declared with the following precondition assertion:

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

We call this precondition assertion *implicit* because it has not been added to the program in the form of an explicit, [P2900R7] style precondition assertion, but is instead implicitly generated by the compiler.

Similarly to signed integer overflow, we can replace any other occurrence of "for operation X, if Y is false, the behaviour is undefined" in the C++ Standard with "operation X has the implicit precondition that Y is true", and treat a violation of such an implicit precondition assertion as a contract violation, in the same way as for explicit precondition assertions.

3.2 Safe fallback behaviour

We can classify all kinds of undefined behaviour in C++ into two types: those for which we can define a useful *safe fallback behaviour* for the error case, and those for which we cannot.

For example, for signed integer addition, 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. If we consider integer addition as a built-in operator as above, we can specify that the safe fallback behaviour is, notionally, what the *function body* of this built-in operator will execute.

As another example, when reading an indeterminate value, a safe fallback behaviour is to instead read some unspecified but valid value. Again, we could be more specific (for example, we could instead read zero) but there is no need to do so.

Note that all cases of undefined behaviour that according to [P2795R5] could be treated as erroneous behaviour could instead be specified as operations with an implicit contract assertion and a safe fallback behaviour which serves as the notional implementation of this operation.

3.3 Adding the `assume` semantic

In many cases, safe fallback behaviours will have significant runtime overhead compared to the status quo. The user might therefore choose to enable them in certain contexts such as debug builds or "hardened" builds for safety-critical applications, but prefer the current C++23 behaviour in other contexts. For example, in many performance-sensitive applications, not optimising on the assumption of no integer overflow is likely to cause an unacceptable performance penalty.

In order for C++ to remain successful for high-performance applications, we therefore need to provide the choice between the safe fallback behaviour and the existing, unsafe behaviour today. We further need to provide this choice separately for different kinds of undefined behaviour. In the Contracts framework, such a choice can be provided by offering different evaluation semantics.

If we want to offer the current C++23 behaviour as an option to avoid unacceptable performance regressions, we need to add a fifth evaluation semantic to the four possible evaluation semantics in [P2900R7]:

— `assume`: do not evaluate the contract predicate; if it would not evaluate to `true`, the behaviour is undefined.

The `assume` semantic is a *non-checking* semantic. Unlike the other non-checking semantic, `ignore`, the `assume` semantic allows the compiler to optimise the program based on the assumption that the predicate of the given contract assertion is always `true`. When applied to implicit contract assertions, the `assume` semantic is therefore equivalent to the current specification for core language undefined behaviour in C++23 and earlier, thereby offering the required backwards-compatibility as an option when selecting the evaluation semantic for a particular kind of implicit contract assertions.

3.4 Mapping evaluation semantics to implementation strategies

We follow the model in [P2900R7] that the selection of evaluation semantics is entirely *implementation-defined*, i.e. any implementation could choose any of these semantics for any operation and still be conforming. That is, no implementation is required to check any implicit contract assertions; however, if it chooses to do so, and any such check fails, we can now reason about the behaviour of the program within the scope of the C++ Standard, and no longer have to treat it as undefined and therefore unsafe.

Using again the signed integer overflow example, we can establish a mapping between the five possible evaluation semantics and different possible implementation strategies for mitigating this particular kind of undefined behaviour:

— The GCC compiler option `-ftrapv`, which aborts the program on signed integer overflow, is a conforming implementation of the `quick_enforce` evaluation semantic;

— A sanitiser which detects signed integer overflow and prints a diagnostic is a conforming implementation of the `enforce` or `observe` evaluation 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` evaluation semantic, which silently executes the safe fallback behaviour;

— The default behaviour in C++23 and earlier, 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` evaluation semantic.

We can construct the same mapping for any kind of undefined behaviour for which we can define a safe fallback behaviour. For example, undefined behaviour due to reading an indeterminate value can be respecified such that reading a value has an implicit precondition assertion that the value is not indeterminate; if the value is indeterminate, the result will be some unspecified value (but no longer undefined behaviour). A conforming C++ implementation can apply any of the five evaluation semantics to this implicit precondition assertion. For example, `enforce` gives a diagnostic followed by termination (a sanitiser detecting this case is a conforming implementation of this semantic), `ignore` silently executes the safe fallback behaviour, i.e. simply reads an unspecified but valid value (setting all uninitialised values to `0` is a conforming implementation of this semantic), while `assume` matches the default in C++23 and before, where reading such a value is undefined behaviour (and the compiler can therefore optimise on the assumption that it never happens).

3.5 Non-ignorable implicit contract assertions

For some undefined behaviour in C++, there is no safe fallback behaviour that makes sense. Examples are dereferencing an invalid pointer and out-of-bounds access into a plain array. We can respecify such undefined behaviour in terms of implicit contract assertions, in the same way as we did before. For example, we can notionally interpret subscripting into an array as the following built-in function:

```
template <typename T, size_t N>
T& subscript(T(&array)[N], size_t index)
pre(index < N) {  // implicit precondition assertion
  return array[index];
}
```

If we are subscripting into a pointer `ptr` to an array `a`, but `sizeof(a)` is not statically known, then the implicit precondition assertion cannot be expressed as above, and instead must be checked with some "magic" function that requires additional instrumentation. Nevertheless, we can reason about such an implicit precondition in the same way:

```
template <typename T>
T& subscript(T* ptr, size_t index)
pre(__magic_bounds_check(ptr, index)) {  // implicit precondition assertion
  return ptr[index];
}
```

In such cases, unlike in our previous integer addition example, if the implicit precondition is checked, and a contract violation is detected, there is no possible safe fallback behaviour that would *not* have undefined behaviour and that we could use to replace the notional function body which dereferences `ptr + index`. In our proposed framework, we call such implicit contract assertions *non-ignorable implicit contract assertions*. Operations that violate memory safety and/or type safety typically fall into this category.

Non-ignorable implicit contract assertions cannot be evaluated with the `ignore` or `observe` semantic; the only options permitted are `quick_enforce`, `enforce`, and `assume`.

We can again construct a mapping of the possible evaluation semantics map to different implementation strategies:

— Clang's proposed[2] `-fbounds-safety` extension, which turns out-of-bounds access into a runtime trap, is a conforming implementation of the `quick_enforce` semantic;

— ASan's behaviour of printing a diagnostic and terminating the process when an out-of-bounds access occurs is a conforming implementation of the `enforce` semantic;

— The default, unsafe behaviour in C++23 and below is a conforming implementation of the `assume` semantic.

The same mapping exists for every other kind of undefined behaviour in C++ without a safe fallback behaviour, when respecified as an operation with a non-ignorable implicit contract assertion.

3.6 Contract-violation handling API

Treating undefined behaviour as a contract violation means that undefined behaviour diagnosed at runtime causes a call to the contract-violation handler when the evaluation semantic is `enforce` or `observe`. On an implementation where these evaluation semantics are supported for implicit contract assertions, and the contract-violation handler is replaceable, we need to specify what happens when the user provides a user-defined contract-violation handler and that handler will be called from the (notional) evaluation of an implicit contract assertion. In particular, we need to define the state of the `contract_violation` object for this case. We can do so with only minor modifications to the library API proposed in [P2900R7].

In [P2900R7], the `contract_violation` class has the following five member functions:

```
class contract_violation {
public:
  const char* comment() const noexcept;
  detection_mode detection_mode() const noexcept;
  contract_kind kind() const noexcept;
  std::source_location location() const noexcept;
  contract_semantic semantic() const noexcept;
};
```

[2] See https://discourse.llvm.org/t/rfc-enforcing-bounds-safety-in-c-fbounds-safety

Below, we discuss the impact of adding implicit contract assertions on each member function.

3.6.1 `comment()` and `location()`

We do not propose any changes to the specification of `comment()` and `location()`. The recommended practice in [P2900R7] is that they return a textual representation of the expression that triggered the contract violation and the source location of the contract violation, respectively. The same is possible for violations of implicit contract assertions.

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 however 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).

3.6.2 `detection_mode()`

We propose that the value returned by `detection_mode()` for violations of implicit contract assertions be unspecified.

Until revision [P2900R4], the Contracts MVP contained a preparatory feature in the form of the enum value `detection_mode::evaluation_undefined_behavior` (added via approving [P2811R7]) for the purpose of handling core language undefined behaviour as a contract violation. In revision [P2900R5], this enum value was removed from the Contracts MVP (via approving [P3073R0]). We do not propose to add this enum value back in, or to modify the current specification of the `detection_mode` enum in any other way.

The name `evaluation_undefined_behavior` is a misnomer in multiple ways. First, after respecifying undefined behaviour as a the violation of an implicit contract assertion, it will no longer be undefined behaviour. Second, checking an implicit contract assertion — unlike for explicit contract assertions — does not necessarily involve a contract predicate that can be evaluated (and some implicit contract assertions cannot be specified in such terms).

Regardless of the name, it does not seem useful to describe all violations of an implicit contract assertion via the same enum value for `detection_mode`. There is a wide range of possible implementations for certain implicit contract assertions: compilers offering a "hardening" mode, sanitisers, architectures implementing pointer protection in hardware such as CHERI, etc. These implementations should be allowed to report different modes of detecting the contract violation via different implementation-defined values for `detection_mode` as this may be useful information for the implementer of the user-defined contract-violation handler.

3.6.3 `kind()`

We propose to add a new enum value `implicit` to the enum `assertion_kind`. This value will be returned from the `kind()` function when the contract violation resulted from an implicit rather than an explicit contract assertion:

```
enum class assertion_kind {
  implicit = 0,
  pre = 1,
  post = 2,
  assert = 3
};
```

We chose the numeric value `0` for the new enum value to emphasise that it does not correspond to any kind of explicit contract assertion (i.e. an assertion that the user wrote).

3.6.4 `semantic()`

We do not propose any changes to the specification of `semantic()`. The same options exist for evaluating implicit contract assertions as for explicit contract assertions: the two evaluation semantics that can result in a call to the contract-violation handler are `enforce` and `observe`.

Note however that non-ignorable implicit contract assertions such as a pointer pointing to a valid object when dereferencing it cannot be evaluated with the `observe` semantic, and therefore `semantic()` will always return `evaluation_semantic::enforce` when violations of implicit contract assertions result in a call to a user-defined contract-violation handler.

4 Discussion

4.1 Making C++ a safer language

The primary aim of this paper is to replace undefined behaviour that is not diagnosable at compile time and instead manifests at runtime with *safe* well-defined behaviour, thereby reducing the overall amount of undefined behaviour in C++ and making it a safer language. We can gradually and iteratively apply the proposed framework for more and more kinds of undefined behaviour in the C++ language, starting with the most obvious candidates.

Candidates for respecification as implicit contract assertions with a *safe fallback behaviour* include reading an indeterminate value (thus replacing [P2795R5]), signed integer overflow, bad bitshifts, and other arithmetic operations and conversions that result in unrepresentable values. In general, all kinds of undefined behaviour that are listed in [P2795R5] as candidates for erroneous behaviour can be respecified in these terms instead.

Candidates for respecification as *non-ignorable* contract assertions (i.e., without a safe fallback behaviour) include all kinds of accessing invalid memory (dereferencing a pointer to an invalid object, out-of-bounds access into an array, etc.), flowing off the end of a non-`void` function, and calling a pure virtual function in an abstract base constructor or destructor.

Note that the choice of evaluation semantic remains implementation-defined and therefore within the purview of the implementation vendor. Compilers that choose to harden certain safety holes such as uninitialised values can document that they use the `ignore` semantic for this particular implicit contract assertion; sanitisers and other tools that add instrumentation to detect memory issues can document that they use the `enforce` semantic for the implicit contract assertions they are designed to diagnose; etc. An implementation that chooses to do nothing at all about undefined behaviour in order to avoid performance regressions (discussed in more detail in Section 4.2) can document that is uses the `assume` semantic for all implicit contract assertions.

The framework proposed here establishes standard vocabulary for all these choices and places their behaviour within the scope of the C++ Standard. This standard vocabulary is useful to reason about the safety of a particular configuration in standard terms. For example, in a safety-critical context, one can now choose to restrict configurations to not use the `assume` the semantic for any implicit or explicit contract assertions, or just for certain kinds of implicit contract assertions that correspond to the largest safety and security risks, and enforce such restrictions on the tooling level.

4.2 A better alternative to erroneous behaviour

This proposal entirely replaces the changes added to the C++26 Working Draft via [P2795R5] and the concept of *erroneous behaviour* introduced by those changes.

Note that when [P2795R5] talks about an operation that *erroneously* does X, this is exactly equivalent to saying that X is the safe fallback behaviour of an operation in case its implicit precondition assertion was violated (regardless of whether this assertion was checked). In both cases, we are talking about the behaviour of a program that is known to be in an incorrect state and that behaviour nevertheless being well-defined. The two terms "erroneous behaviour" and "violation of an implicit contract assertion" to describe such behaviour are therefore essentially interchangeable.

According to [P2795R5], the implementation is permitted to do any of the following when erroneous behaviour occurs:

— Issue a diagnostic;

— Terminate the application;

— Do nothing.

This list of permitted behaviours is a strict subset of the five permitted contract evaluation semantics `quick_enforce`, `enforce`, `observe`, `ignore`, and `assume`. Issuing a diagnostic maps to `observe`; terminating the application with or without such a diagnostic maps to the two enforcing semantics (`enforce` or `quick_enforce`, respectively), and doing nothing maps to `ignore` (silently execute the safe fallback behaviour).

The erroneous behaviour framework contains a subtle difference in its specification of the enforcing semantics. 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 enforcing semantics in [P2900R7] 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.

Note that the erroneous behaviour framework lacks several features of the Contracts framework that can be very useful for handling defects. First, Contracts allow the user to handle the defect programmatically via installing a user-defined violation handler; second, our proposal allows handling of many more kinds of undefined behaviour by also handling cases without a safe fallback behaviour (which we call *non-ignorable* contract violations); and third, our proposal offers the option of full backwards-compatibility with C++23 and earlier via the `assume` semantic.

This last point is particularly significant. Many kinds of undefined behaviour require significant runtime overhead for diagnosis at runtime or even just for changing to a safe fallback behaviour. At the same time, C++ is used in many applications where runtime performance is the absolute highest priority: low-latency trading, video games, real-time signal processing, high-performance computing, etc. Introducing any *measurable* runtime regressions in a new C++ standard will likely lead to such a new standard being rejected by such applications. Continuing to support applications that require maximum performance and cannot afford runtime regressions is as critical for the survival of C++ as improving safety. Any respecification of undefined behaviour to well-defined behaviour that may introduce such runtime regressions therefore requires an escape hatch to revert to the legacy unsafe behaviour of C++23 and earlier, even if the new, safe behaviour is the new default.

The erroneous behaviour [P2795R5] introduces such an escape hatch in the form of a new attribute `[[indeterminate]]` that needs to be added to a variable declaration to get the C++23 behaviour. Since [P2795R5] dooes not offer the `assume` semantic as an option, applying the erroneous behaviour framework to *any* other kind of undefined behaviour will also require such explicit opt-out syntax. This approach does not scale. If we continue to introduce erroneous behaviour to the Standard, we will end up with a whole zoo of various opt-out attributes and other syntactic markers, which will drive up the already high complexity of the language. In addition, the functionality of some of these new syntactic markers, an in particular the fact that they are *unsafe*, might not be obvious to users,

inviting new sources of safety bugs. For other cases such as overflowing arithmetic operations, it is not clear what an opt-out syntax could even look like and whether it is possible. Finally, the cost of correctly applying new opt-out syntax to legacy codebases may be too high for some companies, which means that in order to avoid performance regressions, they will have to permanently remain on older compilers.

By contrast, the `assume` semantic offers a unified mechanism to opt into the unsafe behaviour of C++23 and earlier if needed, without having to change any existing code. This approach avoids unnecessary complexity, is easier to teach, and easier to enforce with tooling, as the unsafe opt-in is entirely contained inside the choice of contract evaluation semantic. Note that the choice of evaluation semantic is implementation-defined, and the granularity of this choice is arbitrary: it can be global, per TU, per kind of undefined behaviour, or even per assertion; compilers are free to offer configuration options for all these cases. If strict safety requirements need to be followed, the usage of the `assume` semantic can be outright banned for both implicit and explicit contract assertions. In addition, a conforming implementation can choose to never use the `assume` semantic by default, and even to not offer this semantic as an option at all.

4.3 Providing a standard C++ API for sanitisers

Several kinds of undefined behaviour that can be respecified in terms of contract violations as proposed here require additional instrumentation to diagnose at runtime. This is particularly true for assertions related to memory safety such as whether a pointer points to a valid object. Diagnosing such issues is the mainstay of sanitisers such as ASan and UBSan. Our proposal provides a standard API that such sanitisers can use to interact with user code.

There is limited existing practice for such APIs. All clang sanitisers have a callback `__sanitizer_set_death_callback`, taking no arguments. This callback 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.

Compared to these callbacks, the library API of the Contracts MVP (with the slight modifications proposed in this paper) provides not only a user callback in the form of a 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, thus placing them much more firmly within the scope of the C++ standard.

Note that supporting this API is entirely optional for existing sanitisers. [P2900R7] specifies that the behaviour of the default contract-violation handler is implementation-defined and an implementation does not have to offer the option of installing a user-defined contract-violation handler, therefore existing sanitisers are already conforming. Any given sanitiser can be said to apply the `enforce` evaluation semantic (without a replaceable contract-violation handler) to those kinds of undefined behaviour that it diagnoses, and the `assume` evaluation semantic to the remaining kinds.

4.4 Preconditions in the C++ standard library

The C++ standard library currently specifies that the violation of any precondition of a standard library function results in undefined behaviour. It follows that many common functions such as `std::vector::operator[]` are inherently unsafe according to the C++ standard.

Vendors that wish to offer safer implementations are already free to strengthen this specification and provide implementations where violations of certain preconditions result in termination or some other well-defined behaviour (e.g. triggering a breakpoint in debug mode). Contracts do not impose

any additional restrictions on library vendors: they can use the new explicit contract assertions as specified in [P2900R7] for this purpose, but are not required to do so.

The concept of implicit contract assertions proposed in this paper extends the concepts of contract violations and evaluation semantics to not only [P2900R7]-style explicit contract assertions, but also preconditions asserted through other means, including compiler built-ins. A standard library implementation can document which of the preconditions in the standard library it implements via explicit contract assertions, which it considers to be implicit contract assertions, and which evaluation semantics it provides for these implicit contract assertions. The framework proposed here would therefore allow vendors to communicate which safety guarantees their standard library implementation provides or does not provide using standard-conforming vocabulary, and place the mitigation strategies employed by these vendors within the scope of the C++ standard, rather than into the realm of vendor-specific extensions. This is relevant for ongoing efforts such as the currently proposed[3] hardening modes for the libc++ standard library implementation.

4.5 Errors leading to `std::terminate` being called

In addition to respecifying undefined and erroneous behaviour to be a contract violation, we can consider respecifying errors that lead to `std::terminate` being called to be a contract violation as well. Such errors are typically unrecoverable situations that arise during exception handling and require abandoning stack unwinding and exception handling in favour of program termination. This approach is explored in more detail in another paper, [P3205R0]; below, we provide a sketch how this approach fits into our proposed framework.

The most common error that leads to `std::terminate` being called is the attempt to throw an exception from a function marked `noexcept`. Handling this error can be integrated into the Contracts framework by specifying that a `noexcept` function has an implicit contract assertion that it will not exit via an exception; thus, an attempt to do so is a contract violation, and the usual contract evaluation semantics apply.

To preserve backwards-compatibility with the behaviour in C++23 and earlier, we can define the call to `std::terminate` to be the safe fallback behaviour[4] for this contract violation (i..e. the behaviour that occurs if the `ignore` semantic is used for this particular contract assertion). In addition, we can opt in to other useful behaviours via the other four semantics. For example, the `enforce` or `observe` semantics offer the options to report the error or even to unwind the stack anyway by using a throwing contract-violation handler, and the `assume` semantic offers the option of more aggressive code optimisations than is possible today[5].

5 Proposed wording

Proposed wording will be added once there is agreement on which specific C++ operations that have the potential of runtime undefined behaviour should be respecified in the fashion proposed here (see Section 4.1 for discussion) and once the Contracts MVP is approved (wording for the respecification proposed here will heavily rely on the wording of the Contracts MVP).

[3] See https://discourse.llvm.org/t/rfc-hardening-in-libc/

[4] Note that we cannot guarantee the backwards-compatible call to `std::terminate` by simply using the enforcing evaluation semantics, because they are not guaranteed to call `std::terminate` specifically, only to terminate the program in some implementation-defined way. Note further that for undefined behaviour, the evaluation semantic that is equivalent to the current C++23 behaviour is the `assume` semantic, while for errors that lead to `std::terminate` being called, the evaluation semantic that is equivalent to the current C++23 behaviour is the `ignore` semantic.

[5] Today, the compiler is only allowed to optimise based on the assumption that an exception will never escape a `noexcept` function because `std::terminate` will be called; with the `assume` semantic, the compiler would additionally be allowed to time-travel-optimise-out all code paths that would lead to such a call to `std::terminate`.

References

[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.

[ONCD2024] The White House. Back to the Building Blocks: A Path Toward Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024-02.

[P1995R1] Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, and Herb Sutter. Contracts – Use Cases. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1995r1.html, 2020-03-02.

[P2687R0] Bjarne Stroustrup and Gabriel Dos Reis. Design Alternatives for Type-and-Resource Safe C++. https://wg21.link/p2687r0, 2022-10-15.

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

[P2811R7] Joshua Berne. Contract-violation handlers. https://wg21.link/p2811r7, 2023-06-27.

[P2900R4] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r4, 2024-01-16.

[P2900R5] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r5, 2024-02-15.

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r7, 2024-05-22.

[P3073R0] Timur Doumler. Remove `evaluation_undefined_behavior` and `will_continue()` from the Contracts MVP. https://wg21.link/p3073r0, 2024-01-26.

[P3205R0] Gašper Ažman, Jeff Snyder, Andrei Zissu, and Ben Craig. Throwing from a `noexcept` function should be a contract violation. https://wg21.link/p3205r0, 2024-04-15.