Making erroneous behaviour compatible with Contracts

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

Document #:	P3229R0
Date:	2025-01-13
Project:	Programming Language C++
Audience:	SG21, EWG, LEWG

Abstract

This paper proposes the first step towards [P3100R1] — a unified framework for describing and handling incorrect C++ programs. In this framework, undefined behaviour, erroneous behaviour, and contract violations are all different aspects of a coherent whole. While most of the changes proposed in [P3100R1] can wait until C++29, this paper contains the part that needs to be adopted for C++26 to avoid setting in stone inconsistencies between the concepts of *erroneous behaviour* on the one hand and *contract violation* on the other hand that would greatly hinder future evolution towards [P3100R1]. Adopting this change paves the way for a brighter future and clarifies the scope of erroneous behaviours already adopted.

1 Motivation and context

C++ is at an inflection point. To effectively address the safety and security challenges facing the C++ ecosystem, a *holistic* strategy is needed. A key part of this strategy is to introduce a unified framework to the C++ Standard that describes *incorrect* programs, i.e. programs whose source code has a bug, and mitigating such bugs during program execution, without leaving the behaviour undefined. This approach is complementary to introducing *static* language constraints that make unsafe constructs ill-formed (e.g., [P3390R0]), and targets bugs that cannot be detected statically.

Such a unified framework is being proposed in [P3100R1]. That proposal is large and the proposed specification is not yet fully complete; it is thus not in scope for C++26. However, there is a small subset of [P3100R1] that must be applied *before* we ship C++26, otherwise we would be setting in stone inconsistencies that would greatly hinder future evolution towards [P3100R1]. This paper proposes to apply just this part to the C++26 Working Draft.

The issue is that we currently have *two* separate specification tools for describing incorrect programs, both of which are independently heading towards C++26: *erroneous behaviour* ([P2795R5]; already merged into the C++26 WD) and *contract violations* ([P2900R13]). Both convey the same concept: at some point during execution, a program is found to be defective, yet its behaviour at that point is still well-defined. Furthermore, both provide essentially the same set of four possible evaluation semantics. According to [intro.abstract] in the current C++26 WD:

> If the execution contains an operation specified as having erroneous behaviour, the implementation is permitted to issue a diagnostic and is permitted to terminate the execution at an unspecified time after that operation.

Note the following equivalence between the options allowed by erroneous behaviour and the contract evaluation semantics proposed in [P2900R13]:

— issuing a diagnostic and terminating is isomorphic to using the *enforce* semantic with the default contract-violation handler;[1]

— terminating without issuing the diagnostic is isomorphic to the *quick-enforce* semantic;

— issuing the diagnostic without terminating is isomorphic to the *observe semantic*;

— opting to not check for erroneous values and let the program continue as-is is equivalent to the *ignore* semantic.

Other than not being hooked into a user-replaceable contract-violation handler, an expression `E` that evaluates to an erroneous value is thus essentially equivalent to the following function:

```
auto eval_E()
post (r: !is_erroneous(r)) {
  return E;
}
```

If we break them down, we can see that the two approaches to handling incorrect programs are very similar in shape and differ primarily in nomenclature:

— *Erroneous behaviour* identifies behaviour that is incorrect (yet well defined) and implies the condition under which that behaviour is reached;

— *Contract assertions* explicitly identify the condition under which the behaviour is considered incorrect, and leave it up to the user to implement the — possibly well-defined but possibly undefined — behaviour that follows a violation.

From that description, it is clear that any case of well-defined behaviour after a contract violation is nothing other than erroneous behaviour. Similarly, any condition that identifies erroneous behaviour as part of an operation is nothing other a contract assertion on that operation. The only difference is that erroneous behaviour is always *well-defined* behaviour that follows a contract violation, while contract violations can in principle also be followed by undefined behaviour (if *ignored* or *observed*).

Understanding now that erroneous behaviour is a specific category of behaviour that might follow a contract violation, we must then consider the concrete differences in the behaviours specified for erroneous in the draft Standard today and how a contract violation followed by matching well-defined behaviour would behave:

— While printing a diagnostic message is recommended behaviour for both contract violations and erroneous behaviour, [P2900R13] allows the user to override this behaviour and install their own user-defined contract-violation handler, while erroneous behaviour does not.

— When a contract assertion is evaluated with a terminating semantic (*enforce* or *quick-enforce*) and a contract violation has been detected, [P2900R13] specifies a set of conforming modes of termination,[2] while erroneous behaviour does not.

[1]Strictly speaking, the effects of the default contract-violation handler are implementation-defined; in theory, an implementation could choose to do anything it wants. However, the recommended practice in [P2900R13] is that the default contract-violation handler prints a diagnostic message. We expect this recommended practice to be followed on all major compilers. Note that this behaviour is also backwards-compatible with the default behaviour of the C `assert` macro.

[2]The conforming modes of termination on contract violation are the three erroneous termination modes available in C++: `std::terminate`, `std::abort`, and immediate termination, such as via a trap instruction. All three have important use cases; detailed discussion can be found in [P3520R0].

— If the program is terminated due to a contract violation, termination happens as part of evaluating the contract assertion and immediately after the violation has been handled; on the other hand, if erroneous behaviour is encountered, termination happens "at an unspecified time after that operation", thereby introducing some sort of "Damocles semantic".

— If an expression would evaluate an operation that would have erroneous behaviour, it does not qualify as a core constant expression, which in turn allows the user to SFINAE on whether an expression results in erroneous behaviour. On the other hand, [P2900R13] does not allow the user to SFINAE on the presence or evaluation of a contract assertion: if an *enforced* contract assertion fails during constant evaluation, the program is straight up ill-formed.

To remove the above inconsistencies and pave the way towards a unified standard framework for reasoning about incorrect programs, we need to change the terminology and specification of erroneous behaviour to align with Contracts — *before* shipping erroneous behaviour in C++26.

2 Proposed design

2.1 Implicit contract assertions

The first and most important step towards the unified framework proposed in [P3100R1] is to extend the notion of *contract assertion* by distinguishing between *explicit* and *implicit* contract assertions. Explicit contract assertions are those that [P2900R13] proposes. They are added to the program explicitly using function contract specifiers (`pre`, `post`) and assertion statements (`contract_assert`). By contrast, implicit contract assertions are not directly visible in code, but injected into the program by the implementation. Otherwise, explicit and implicit contract assertions behave the same. Importantly, they can be evaluated with the same four evaluation semantics — *ignore*, *observe*, *enforce*, and *quick-enforce* — and call the same global contract-violation handler.

The second step is to recognise that the occurrence of erroneous behaviour — such as when a builtin operation results in an erroneous value — is equivalent to the violation of an implicit contract assertion that the operation does *not* result in an erroneous value. We can then remove the notion of erroneous behaviour from the Standard specification entirely and instead say that the evaluation of any builtin operation includes an implicit postcondition assertion that it does not produce an erroneous value. Following [P2900R13], that implicit postcondition assertion can then be evaluated with any of the four evaluation semantics, a violation can trigger the contract-violation handler, etc. This modification instantly removes all inconsistencies described in Section 1.

Calling a user-defined contract-violation handler on erroneous behaviour is already existing practice: UBSan, an implementation that detects erroneous behaviour, offers an API for installing such a custom handler.[3] As discussed in [P3100R1], there are many benefits in standardising an API for such callbacks via the replaceable contract-violation handler, and allowing any program defect detected during program execution, including erroneous behaviour, to trigger that same handler.

The same strategy can be applied to all other cases where the implementation determines at runtime that the program code has a bug. This includes all occurrences of undefined behaviour which [P2795R5], Section "The Broader Picture" lists as a "candidate for conversion to erroneous behaviour": signed integer overflow, unrepresentable arithmetic conversions, bad bitshifts, and so on. We can remove the undefined behaviour from all these operations by specifying the appropriate implicit precondition or postcondition assertions and a well-defined fallback behaviour; see [P3100R1]. It also includes implicit contract violations inserted by Profiles; see [P3081R1].

[3]All Clang sanitisers offer the API `__sanitizer_set_death_callback` for this purpose; ASan additionally offers a slightly more sophisticated API `__asan_set_error_report_callback`.

Note that — apart from the issue discussed in Section 2.3 — the change proposed here is *non-breaking*: a conforming implementation of [P2795R5] will remain conforming with this paper. Note further that while we propose to rephrase erroneous behaviour entirely in terms of implicit contract assertions and thus remove the notion of *erroneous behaviour* from the C++ Standard, the notion of *erroneous value* remains useful, and in fact we propose below that it should be extended in order to more clearly achieve its original goals.

2.2 Extending the contract-violation handling API

In order to introduce implicit contract assertions on top of [P2900R13], we need to extend the contract-violation handling API, which currently only covers explicit contract assertions. Specifically, we need to specify the state of the `std::contracts::contract_violation` object that a user-defined contract-violation handler will see when such an implicit contract assertion is violated. Only two small extensions to that API are needed.

The first extension concerns the `kind()` property of `contract_violation`, which returns an enum value that represents the syntactic form of the violated contract assertion. In addition to the three enumerators `pre`, `post`, and `assert` from [P2900R13], which represent explicit precondition assertions, postcondition assertions, and assertion statements, respectively, we propose a fourth enumerator `implicit`, which represents all forms of implicit contract assertions, including the ones resulting from erroneous values that we propose to introduce here. Note that the same enumerator is being proposed in [P3100R1] (of which this paper is a subset) and also in [P3081R1] (Profiles).

The second extension concerns the `detection_mode()` property of `contract_violation`, which returns an enum value that represents the particular failure mode. In addition to the three enumerators `predicate_false` and `evaluation_exception` from [P2900R13], we propose a third enumerator `erroneous_value`, which represents a contract violation due to an erroneous value being produced. Note that this is distinct from the evaluation of a boolean predicate returning `false` or exiting via an exception, because the determination that an erroneous value was produced (and therefore a contract violation occurred) takes place in an implementation-defined fashion and not necessarily by evaluating a boolean predicate; therefore, the existing two enumerators do not apply. This is again fully consistent with the direction in [P3100R1] as well as with [P3081R1], which introduces its own values for `detection_mode()` for the new failure modes added by Profiles.

The numeric values of the enumerators do not carry any special meaning (see [P3327R0], Section 3.4 for rationale), so it does not matter very much how we choose them. Here, we propose to make `assertion_kind::implicit` first in the list (value `1`), because implicit contract assertions are the only ones that are *not* introduced by a syntactic specifier and thus are distinct from all other kinds of assertions, including kinds we might add in C++29 or later, such as class invariants and procedural interfaces. However, if a breaking change to [P2900R13] is not desired, we might as well append it to the list (value `4`). As for `detection_mode`, any proposal adding a new detection mode to the Contracts framework, including this one, should simply append them to the list.

2.3 Replacing the “Damocles semantic” by “sticky” erroneous values

With erroneous behaviour as specified today, a program effectively enters an “erroneous state” once any erroneous behaviour happens, and the program may be unceremoniously terminated at any point after that. The expectation is that it will be either reasonably soon or never, but the specification gives significantly more freedom to the implementation, making it much harder to reason about the behaviour of the program.

We, as a community, are lucky enough to already have implementations that identify and trap on uses of uninitialised values. The issue, however, is that the implementation of such checks is often done as instrumentation of compiled code, and thus can be a less than direct mapping of reading of C++ expression that does a first uninitialised read to the instructions that can be turned into a diagnostic error of some kind. In practice, it is not the read of the uninitialised value that is identified but rather the first case where a branch is taken based on that value — and this can be much later than the actual read. This implementation strategy is the original motivation for the current specification of erroneous behaviour as introduced by [P2795R5].

On the other hand, such implementations will never notice an uninitialised read and then set a timer, wait an arbitrary time, and terminate the program while it is doing something completely unrelated. The specification of erroneous behaviour today allows this choice unnecessarily.

To fix this issue while retaining compatibility between the specification and existing implementations, we propose that erroneous values be "sticky" in a way that they are not today. The specification today takes the approach that once you read an erroneous value the value itself is then "cleaned" and no longer toxic, but comes with the downside that all following program behaviour is now erroneous and thus at risk of termination. By having erroneous values propagate with the data, however, we can extend the scope of the problematic data sufficiently to cover all realistic implementations while avoiding the more toxic threat of eventual and surprising termination at a later date.

In practice, we do not expect the erroneous of a value to transport beyond a single function or translation unit, but we should still allow for that possibility if an adventurous platform chooses to do so. Therefore, we simply make any operation whose result is dependent on an erroneous value produce an erroneous value. Each such operation will fail its implicit postcondition assertion that the value produced is not erroneous, but as per the [P2900R13] model of implementation-defined choice of evaluation semantic, the implementation is free to evaluate any of these implicit postcondition assertions with the *ignore* semantic. A program is therefore free to discard all such data and move on. Such a program is now no longer under threat of unexpected termination, while leaving us maximal flexibility for identifying the bug when it is most convenient to do so.

2.4 Interaction with `noexcept`

Consider:

```
bool f() {
  int x;
  return noexcept(x + 1);
}
```

In C++ today, calling `f()` has defined behaviour (the indeterminate value is never accessed; the operand of `noexcept` is an unevaluated operand) and returns `true` (adding two integers can never throw an exception unless the behaviour is undefined). If we want to avoid breaking changes to the existing language, the result of the `noexcept` operator must remain the same with this proposal.

However, since `x` has an erroneous value, evaluating `x + 1` may call the contract violation handler, which may throw an exception. With this proposal, it is therefore no longer true that `x + 1` can never throw an exception unless the behaviour is undefined.

A detailed discussion of this problem can be found in [P3541R1]. The possible solutions broadly fall into three categories: either accept the breaking change to the `noexcept` operator, or do not allow throwing violation handlers for implicit contract assertions, or redefine the meaning of the `noexcept` operator to be "can never throw an exception *unless there is a contract violation*". SG21 discussed this problem at great length and in the end achieved consensus that the only acceptable solution is the last one. We follow this decision in this paper.

It is therefore possible for an implicit contract assertion to call a throwing contract-violation handler when violated, and for the evaluation of the expression to exit via that exception, even if the `noexcept` operator returns `true` for that expression.

Note that this design is fully consistent with [P3081R1], which proposes the same behaviour for the implicit contract assertions inserted by Profiles.

2.5 Constant evaluation

The change proposed here makes it impossible to SFINAE on whether an expression results in an erroneous value. Such compile-time branching on the presence of a bug should never be allowed; a detailed rationale can be found in [P2900R13] Section 3.1, which enshrines this design principle in the so-called *Contracts Prime Directive*.

In [P2900R13], when an explicit contract assertion is encountered during constant evaluation, it can be *ignored* (the predicate is not evaluated at all), *observed* (if the predicate does not evaluate to `true`, a compiler warning is issued), or *enforced* (if the predicate does not evaluate to `true`, the program is ill-formed).

Explicit contract assertions are user-authored and have important use cases for all three evaluation semantics. The *ignore* semantic is useful in code bases where constant evaluation of all contract predicates would have a prohibitively costly impact on compile time, while the *observe* semantic is useful when the author is not yet confident about the correctness of the check itself.

On the other hand, implicit contract assertions do not fall in either category. We can therefore remove the possibility to SFINAE on whether an expression results in an erroneous value while preserving the property of the current C++26 WD that during constant evaluation, an erroneous value can never be produced. We can achieve this by specifying that during constant evaluation, implicit contract assertions can only be evaluated with the *enforce* semantic. In short, with our proposal, if any implicit contract assertions fails during constant evaluation, the program is straight up ill-formed.

3 Proposed wording

The wording changes proposed in this section are relative to the C++ Working Draft [N5001] with the changes proposed in [P2900R13] (Contracts) already applied. The proposed changes are also designed to be compatible with [P3081R1] (Profiles).

Replace “contract assertion” with “explicit contract assertion” in all places in the wording that do not apply to implicit contract assertions. This complete list will be reproduced when a core-approved version of the wording in [P2900R13] is available.

Remove [defns.erroneous]:

> ~~**erroneous behavior**~~
>
> ~~well-defined behavior that the implementation is recommended to diagnose~~
>
> ~~[*Note to entry:* Erroneous behavior is always the consequence of incorrect program code. Implementations are allowed, but not required, to diagnose it ([intro.compliance.general]). Evaluation of a constant expression ([expr.const]) never exhibits behavior specified as erroneous in Clause 4 through Clause 15. — *end note*]~~

Modify [intro.compliance.general], footnote 3:

> “Correct execution” can include undefined behavior and ~~erroneous behavior~~contract violations ([basic.contract]), depending on the data being processed; see Clause 3 and [intro.execution].

Modify [intro.abstract]:

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation). ~~If the execution contains an operation specified as having erroneous behavior, the implementation is permitted to issue a diagnostic and is permitted to terminate the execution at an unspecified time after that operation.~~

~~*Recommended practice:* An implementation should issue a diagnostic when such an operation is executed.~~

~~[*Note:* An implementation can issue a diagnostic if it can determine that erroneous behavior is reachable under an implementation-specific set of assumptions about the program behavior, which can result in false positives. — *end note*]~~

Modify [basic.contract.general] as follows:

Contract assertions ~~allow the programmer to~~ specify properties of the state of the program that are expected to hold at certain points during execution.

Explicit c~~C~~ontract assertions are introduced by *precondition-specifier*s, *postcondition-specifier*s ([dcl.contract.func]), and *assertion-statement*s ([stmt.contract.assert]). *Implicit* contract assertions are inserted into the program by the implementation.

Each contract assertion has a *predicate*, which is an expression of type `bool`. ~~[*Note:* The value of the predicate is used to identify program states that are expected. — *end note*]~~ If it is determined during program execution that the predicate has a value other than `true`, a *contract violation* occurs. A contract violation is always the consequence of incorrect program code.

Modify [basic.contract.eval]:

An evaluation of a contract assertion uses one of the following four *evaluation semantics*: *ignore*, *observe*, *enforce*, or *quick-enforce*. Observe, enforce, and quick-enforce are checking semantics; enforce and quick-enforce are terminating semantics.

Which evaluation semantic is used for any given evaluation of a contract assertion is implementation-defined. During constant evaluation, implicit contract assertions are always evaluated with the enforce semantic. [*Note:* The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

Recommended practice: An implementation should provide the option to translate a program such that all evaluations of explicit contract assertions use the ignore semantic as well as the option to translate a program such that all evaluations of explicit contract assertions use the enforce semantic. By default, evaluations of explicit contract assertions should use the enforce semantic.

Modify [basic.indet] as follows:

When storage for an object with automatic or dynamic storage duration is obtained, the bytes comprising the storage for the object have the following initial value:

— If the object has dynamic storage duration, or is the object associated with a variable or function parameter whose first declaration is marked with the `[[indeterminate]]` attribute ([dcl.attr.indet]), the bytes have *indeterminate values*;

— otherwise, the bytes have *erroneous values*, where each value is determined by the implementation independently of the state of the program.

If no initialization is performed for an object (including subobjects), such a byte retains its initial value until that value is replaced ([dcl.init.general,expr.ass]). If any bit in the value representation has an indeterminate value, the object has an indeterminate value; otherwise, if any bit in the value representation has an erroneous value, the object has an erroneous value ([conv.lval]).

[*Note:* Objects with static or thread storage duration are zero-initialized, see ([basic.start.static]). — *end note*]

The evaluation of any builtin operation includes an implicit postcondition assertion that it does not produce an erroneous value. Except in the following cases, if any operand of a built-in operator is erroneous then the value produced by that operator is erroneous:

- If the built-in operation is `*` or `&` and the other operand is a non-erroneous zero value.
- If the built-in operation is `|` and the other operand is a non-erroneous value whose base-2 representation all of whose coefficients are `1`.

Except in the following cases, if an indeterminate value is produced by an evaluation, the behavior is undefined and if an erroneous value is produced by an evaluation ~~, the behavior is erroneous and~~ the result of the evaluation is the value so produced ~~but is not erroneous~~ and it is erroneous:

- If an indeterminate or erroneous value of unsigned ordinary character type ([basic.fundamental]) or `std::byte` type ([cstddef.syn]) is produced by the evaluation of:
  - the second or third operand of a conditional expression ([expr.cond]),
  - the right operand of a comma expression ([expr.comma]),
  - the operand of a cast or conversion ([conv.integral, expr.type.conv,expr.static.cast,expr.cast]) to an unsigned ordinary character type or `std::byte` type ([cstddef.syn]), or
  - a discarded-value expression ([expr.context]),

  then the result of the operation is an indeterminate value or that erroneous value, respectively.
- If an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator ([expr.ass]) whose first operand is an lvalue of unsigned ordinary character type or `std::byte` type, an indeterminate value or that erroneous value, respectively, replaces the value of the object referred to by the left operand.
- If an indeterminate or erroneous value of unsigned ordinary character type is produced by the evaluation of the initialization expression when initializing an object of unsigned ordinary character type, that object is initialized to an indeterminate value or that erroneous value, respectively.
- If an indeterminate value of unsigned ordinary character type or `std::byte` type is produced by the evaluation of the initialization expression when initializing an object of `std::byte` type, that object is initialized to an indeterminate value or that erroneous value, respectively.

Converting an indeterminate or erroneous value of unsigned ordinary character type or `std::byte` type produces an indeterminate or erroneous value, respectively. In the latter case, the result of the conversion is the value of the converted operand.

Modify [expr.const], paragraph 10:

An expression E is a *core constant expression* unless the evaluation of E, following the rules of the abstract machine ([intro.abstract]), would evaluate one of the following:

...

— an operation that would have undefined ~~or erroneous~~ behavior as specified in Clause 4 through Clause 15;

...

Modify [dcl.attr.indet], paragraph 3:

[*Note:* Reading from an uninitialized variable that is marked `[[indeterminate]]` can cause undefined behavior.

```
void f(int);
void g() {
  int x [[indeterminate]], y;
  f(y);          // ~~erroneous behavior~~contract violation ([basic.indet])
  f(y);          // undefined behavior
}

[...]
```

— *end note*]

Modify [nullablepointer.requirements], paragraph 2:

A value-initialized object of type `P` produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type `P` may have an indeterminate or erroneous value.

Note: Operations involving indeterminate values can cause undefined behavior, and operations involving erroneous values can cause ~~erroneous behavior~~contract violations ([basic.indet]).— *end note*]

Modify [bit.cast], paragraph 2:

For each bit *b* in the value representation of the result that is indeterminate or erroneous, let *u* be the smallest object containing that bit enclosing *b*:

— If *u* is of unsigned ordinary character type or `std::byte` type, *u* has an indeterminate value if any of the bits in its value representation are indeterminate, or otherwise has an erroneous value.

— Otherwise, if *b* is indeterminate, the behavior is undefined.

— Otherwise, ~~the behavior is erroneous~~an implicit contract violation ([basic.contract]) occurs, and the result is as specified above.

The result does not otherwise contain any indeterminate or erroneous values.

Modify [contracts.syn]:

```
// all freestanding
namespace std::contracts {

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

```
[...]

enum class detection_mode :  unspecified
  predicate_false = 1,
  evaluation_exception = 2,
  erroneous_value = 3
};
```

Modify [support.contracts.enum.kind]:

> The enumerators of `assertion_kind` correspond to the possible syntactic forms of a contract assertion ([basic.contract]):
>
> — `assertion_kind::implicit`: the evaluated contract assertion was an implicit contract assertion.
>
> — `assertion_kind::pre`: the evaluated contract assertion was an explicit precondition assertion.
>
> — `assertion_kind::post`: the evaluated contract assertion was an explicit postcondition assertion.
>
> — `assertion_kind::assert`: the evaluated contract assertion was an explicit assertion-statement.

Modify [support.contracts.enum.detection]:

> The enumerators of `detection_mode` correspond to the manners in which a contract violation ([basic.contract.eval]) can be identified:
>
> — `detection_mode::predicate_false`: the contract violation occurred because the predicate evaluated to `false` or would have evaluated to `false`.
>
> — `detection_mode::evaluation_exception`: the contract violation occurred because the evaluation of the predicate exited via an exception.
>
> — `detection_mode::erroneous_value`: the contract violation occurred because an erroneous value was encountered.

Bibliography

[N5001] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/n5001.pdf, 2024-12-17.

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

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r13, 2025-01-13.

[P3081R1] Herb Sutter. Core safety profiles for C++26. https://wg21.link/p3081R1, 2025-01-06.

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne. Undefined and erroneous behaviour is a contract violation. https://wg21.link/p3100r1, 2024-10-16.

[P3327R0] Timur Doumler. Contract assertions on function pointers. https://wg21.link/p3327r0, 2024-09-17.

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

[P3520R0] Timur Doumler, Joshua Berne, and Andrzej Krzemieński. Contracts for C++: Wrocław technical fixes. https://wg21.link/p3520r0, 2024–11-21.

[P3541R1] Andrzej Krzemieński. Violation handlers vs noexcept. https://wg21.link/p3541r1, 2025-01-06.