Fix erroneous behaviour termination semantics for C++26

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)

Document #: P3684R1
Date: 2025-11-07
Project: Programming Language C++
Audience: EWG, CWG

Abstract

We propose a small adjustment to the specification of erroneous behaviour in C++26. The current specification gives the implementation freedom to terminate the program at any arbitrary point after erroneous behaviour has occurred. This specification is an impediment to applying the concept of erroneous behaviour more broadly across the C++ language, and prevents consistency in semantics between erroneous values and other types of implicit contract violations. We introduce an improved specification for program termination following erroneous behaviour that removes these impediments while retaining full compatibility with existing implementations.

1 Motivation and context

C++26 introduces the concept of *erroneous behaviour*: behaviour that is known to be incorrect yet is not undefined. It also introduces one concrete instance of erroneous behaviour: in C++26, reading a default-initialised variable of automatic storage duration and scalar type now produces an erroneous value instead of an indeterminate value; where using an indeterminate value has undefined behaviour, use of an erroneous values instead has erroneous behaviour. This removal of undefined behaviour is designed to mitigate existing (and actively exploited) security vulnerabilities.

Beyond that one instance of erroneous behaviour, [P2795R5] contains a section with a tentative list of other cases of undefined behaviour that could be replaced with erroneous behaviour in the future. [P2973R0] proposes to do so concretely for the case of a missing return from an assignment operator. [P3100R4] goes further and proposes to replace undefined behaviour with erroneous behaviour systematically across the entire C++ Standard for all cases in which plausible replacement behaviour exists; the paper identifies 16 such cases.

The core language UB white paper [P3656R1] currently pursued by EWG recognises erroneous behaviour as one of three key tools to address the issues caused by undefined behaviour in C++ (Contracts and Profiles being the other two). [P3229R0] points out the isomorphism between the semantics of erroneous behaviour and those of well-defined code following a failed contract assertion. That paper proposes a path towards making erroneous behaviour and implicit contract assertions fully consistent and synergetic with each other.

Given the important role of erroneous behaviour for the future evolution of C++, we should ensure that the specification we ship in C++26 is forward-compatible with a broader application of this concept, beyond the immediate benefit of some uninitialised reads no longer being undefined.

2 The problem

In the current C++26 working draft [N5014], the evaluation semantics of erroneous behaviour are specified as follows ([intro.abstract]):

> 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.

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

At the time of writing, two flavours of [P2795R5] implementations exist. The first strategy is to initialise default-initialised variables of automatic storage duration and scalar type to a fixed repeating pattern or to zero. Clang and GCC offer this functionality via the flag `-ftrivial-auto-var-init`. This strategy removes the undefined behaviour but with the caveat that it never exhibits the encouraged behaviour of diagnosing an error.

The second strategy is to actually detect such erroneous reads. This is implemented by MemorySanitizer, which terminates with a diagnostic message some time after an erroneous read happens. As the mapping of code in C++ to post-optimisation machine instructions can often be unintuitive, the implementation of such checks is typically done as instrumentation of compiled code. In particular, the exact place where an erroneous read occurs can be rolled into many subsequent operations, and it is those subsequent operations where a diagnostic could potentially be emitted, not the initial read. The case that can actually be caught is often a branch that occurs whose condition is dependent on an uninitialised value, and that branch might be very far removed from where the first read of the uninitialised value occurs in the abstract machine. This implementation strategy is the original motivation for the "delayed termination" that currently features in the specification for erroneous behaviour.

MemorySanitizer itself does not, however, delay identification of erroneous behaviour indefinitely. Instead, because it is detecting branches that are dependent on erroneous bits it will terminate a program only during some evaluations that make use of values that were read from uninitialised storage. Therefore, to leave the behaviour of MemorySanitizer as a conforming implementation we do not need a blanket permission to terminate a program at any time after an erroneous operation has taken place. MemorySanitizer actually tracks this dependence on uninitialised values at the bit level, focusing on trapping in only those evaluations that have behaved differently based on erroneous values.

In our expectation, no real-world implementation will detect any form of erroneous behaviour, set a timer, wait an arbitrary time, and then terminate the program while it is doing something completely unrelated. Such an approach to notifying users of a bug would lead to undiagnosable problems in a scenario where we are actively striving to increase the predictability and reliability of our software even in the face of incorrect behaviour. The current specification of erroneous behaviour allows this choice unnecessarily.

Further, there are cases of undefined behaviour that are unrelated to reading unitialised values but can also be replaced with erroneous behaviour in future versions of C++ (see [P3100R4]). In these cases, delayed termination serves no purpose at all. Allowing delayed termination actively prevents establishing consistency in semantics between erroneous values and other types of implicit contract violations (see [P3229R0]).

3 Proposal

We propose to update the specification of erroneous behaviour and erroneous values in a way that achieves the following key properties:

— Existing implementations that trap on the use of uninitialised values, such as MemorySanitizer, will remain conforming implementations and retain the ability to trap when they do with no need for modifications.

— When a program terminates because of erroneous behaviour we will have the ability to reason locally about why it terminated, without needing to identify all possible previous executions that might have been erroneous.

— When a program is in a running state and *not* performing any erroneous operations we can reliably count on it not terminating due to earlier misbehaviours.

To achieve these properties, we propose to remove the "delayed termination" aspect from the specification and instead to make erroneous values "sticky". The current specification takes the approach that once you read an erroneous value, the value itself is "cleaned" and no longer toxic, but that comes at the cost of putting your entire program in what is essentially a toxic state. By having erroneous values instead propagate with the data, 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 time.

In practice, we do not expect the erroneous-ness of a value to transport beyond a single function or translation unit, but we should still allow for that possibility if a platform chooses to do so. Therefore, we make any operation whose result is dependent on an erroneous value produce an erroneous value. Since any such operation is erroneous behaviour, the implementation is free to either issue a diagnostic and/or terminate the program at that point, or to just move on (as neither the diagnostic nor the termination are normatively required). Such a program is now no longer under threat of unexpected termination, while leaving maximal flexibility for detecting the bug when it is most convenient to do so.

For built-in operations that use an erroneous value but whose result does not actually depend on that value — multiplying by 0, mod-ing by 1, and similar cases — we can produce a "clean" non-erroneous value. This would, however, only be sound when the result of an operation is guaranteed to not be dependent on the particular erroneous value being used. For floating-point values there are almost no operations that have this property, as the erroneous value could always be a NaN (binary operations with a non-erroneous NaN will always result in a NaN, but this does not seem like a useful case to give special treatment). For integral values on most platforms we can state results more definitively, but the standard allows for integers that have trapping values or similar unexpected behaviour, so even then we cannot definitively say that any operations have a result that is completely independent of the value of one of its operands.

On the other hand, any platform can choose to treat erroneous values as non-erroneous under any condition — our specification does not put any strict requirements the erroneousness of a value can be detected within the abstract machine. Therefore implementations will always be free to improve their quality by identifying values that are truly independent of erroneous inputs and not terminating when those values are used.

In the future, the Standard can also further narrow the set of values that are treated as erroneous by identifying cases where the resulting value is completely independent of an erroneous operand. For example, if we have an expression that meets all of the following conditions, we could consider making the resulting zero value non-erroneous (and MemorySanitizer effectively does):

— The operator is multiplication (`*`).

— The result type is an integer type.

— One operand is an erroneous value of a type that has no NaN or trapping values.

— The other operand is a non-erroneous zero value.

Similar rules could be written when using the mod (`%`) operator with a 1 value as the righthand argument, using bitwise operators with 0 and all-1 operands, or various bit-shift operations. On the other hand, these rules are complicated, not universally portable, and do nothing but make questionable code slightly less questionable. Therefore we recommend, at least initially, having all expressions that make use of an erroneous value result in an erroneous value.

One subtlety to note is how this interacts with the short-circuiting logical operators. These do not even evaluate their second operand if the first operand determines their result. For these operators we will never actually evaluate a short-circuited subexpression to even know if its values is erroneous, and thus we need to restrict our propagation of erroneous values to come only from those operands that are themselves actually evaluated.

This proposal resolves NB comment GB 02-036.

4 Wording

The proposed wording is relative to [N5014].

Modify [intro.abstract] paragraph 6 as follows:

> A conforming implementation executing a well-formed program shall produce the same observable behavior of the defined prefix of one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. If the selected execution contains an undefined operation, the implementation executing that program with that input may produce arbitrary additional observable behavior afterwards. If the execution ~~contains~~of an operation is 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~~of the program.

Modify [basic.indet], paragraph 2 as follows:

> If any operand of a built-in operator that produces a prvalue is evaluated, is not a discarded-value expression ([expr.context]), and produces an erroneous value, then the value produced by that operator is erroneous.
>
> 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~~that erroneous value:
>
> — 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:
> ...

Acknowledgements

Thanks to Richard Smith, Thurston Dang, Thomas Köppe, and Iain Sandoe for providing valuable feedback.

References

[N5014] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://wg21.link/n5014, 2025-08-05.

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

[P2973R0] Jonathan Wakely and Thomas Köppe. Erroneous behaviour for missing return from assignment. https://wg21.link/p2973r0, 2023-09-15.

[P3100R4] Timur Doumler and Joshua Berne. A framework for systematically addressing undefined behaviour in the C++ Standard. https://wg21.link/p3100r4, 2025-08-13.

[P3229R0] Timur Doumler and Joshua Berne. Making erroneous behaviour compatible with Contracts. https://wg21.link/p3229r0, 2025-01-13.

[P3656R1] Herb Sutter and Gašper Ažman. Initial draft proposal for core language UB white paper: Process and major work items. https://wg21.link/p3656r1, 2025-03-23.