Remove `evaluation_undefined_behavior` and `will_continue()` from the Contracts MVP

Timur Doumler (papers@timur.audio)
Tom Honermann (tom@honermann.net)
Ville Voutilainen (ville.voutilainen@gmail.com)

Document #: P3073R0
Date: 2024-01-26
Project: Programming Language C++
Audience: SG21

Abstract

This paper proposes to remove the enum value `evaluation_undefined_behavior` and the function `will_continue()` from the Contracts MVP. Given the remaining open questions surrounding these two library facilities, and given that neither is essential in order to use the Contracts MVP, or depended upon by any other part of the Contracts MVP, we believe that they should instead be standardised as post-MVP extensions, to give SG21 enough time to resolve these open questions without putting at risk the SG21 roadmap to get the Contracts MVP into C++26.

1 Introduction

By approving [P2811R5], SG21 adopted a standard library API for the Contracts MVP proposal [P2900R4] that allows to query the properties of a runtime contract violation inside a user-defined contract-violation handler invoked upon detection of such a violation. This query happens via an object of type `std::contracts::contract_violation` created by the implementation and passed into the contract-violation handler. For this purpose, `std::contracts::contract_violation` offers a number of property functions, including:

— `detection_mode()`: returns an enum representing the mode by which the contract violation was detected (evaluation of the predicate returned `false` or would have returned `false`; or the evaluation of the predicate exited via an exception);

— `kind()`: returns an enum representing the kind of contract that was violated (precondition, postcondition, or assertion);

— `semantic()`: returns an enum indicating which contract semantic was used to check the contract assertion (*enforce* or *ignore*); and

— `will_continue()`: returns a `bool` value indicating whether evaluation of the program will continue after the current contract-violation handler has returned normally, or whether the program will be terminated.

Recent discussions on the SG21 reflector have revealed that for two components of this proposed standard library API, there is currently a substantial amount of open questions around their meaning, their correct specification, and their motivating use cases. These components are: the enum value `evaluation_undefined_behavior`, which is one of the possible values returned by `detection_mode()`, and the `will_continue()` function. Neither facility is required in order to use the rest of the `std::contracts::contract_violation` API or indeed the Contracts MVP as a whole. We therefore propose, with this paper, that both facilities be removed from the Contracts MVP.

Note that we are *not* suggesting that either facility is not useful or should not be standardised. We merely propose that given the remaining open questions surrounding these two facilities, and given that neither is essential in order to use the Contracts MVP, or depended upon by any other part of the Contracts MVP, they should be treated as post-MVP extensions. Treating these facilities as post-MVP extensions, in the same way as other useful features such as support for virtual functions [D3097R0] and postcondition captures [D3098R0], will give SG21 enough time to resolve the open questions surrounding these facilities without putting at risk the roadmap agreed to by SG21 [P2695R1] and the chances of getting the Contracts MVP into C++26.

2 `evaluation_undefined_behavior`

2.1 What is `evaluation_undefined_behavior` for?

[P2811R5], the proposal approved by SG21 for the Contracts MVP, specifies that returning the enum value `evaluation_undefined_behavior` from `detection_mode()` means that the "contract predicate would have undefined behaviour when evaluated".

Since approval of the paper, it has been suggested that the value could actually mean more than that. In particular, the suggestion is that if a sanitiser or similar tool is capable of detecting undefined behaviour at runtime anywhere in the program and report this to the user, it could do so by calling the contract-violation handler, in which case `std::contracts::contract_violation` would serve as a standard API for such tools to report intercepted defects. In this scenario, `evaluation_undefined_behavior` would signal to the user that the nature of the intercepted defect is an expression that, when evaluated, would have undefined behaviour.

Discussion of this idea on the SG21 reflector revealed that there is no consensus about whether `evaluation_undefined_behavior` should mean that:

— Evaluation of the contract predicate expression itself, or one of its immediate subexpressions, would have undefined behaviour, or

— Evaluation of any expression while evaluating the contract predicate, including inside any function (possibly in another translation unit) called during such evaluation, would have undefined behaviour, or

— Evaluation of any expression anywhere during the execution of the program would have undefined behaviour, regardless of whether the program actually contains any contract assertions.

2.2 Is `evaluation_undefined_behavior` the correct API for runtime detection of undefined behaviour?

We believe that adding a standard API to C++ for sanitisers and other tools to report undefined behaviour to the user is a very promising idea. This approach, if done right, has the potential to integrate three different strategies to mitigate undefined behaviour in C++ — contract-violation handling, runtime detection of core language undefined behaviour, and the new concept of *erroneous*

behaviour currently in development (see [P2795R4] and [P2973R0]) — into a single integrated facility designed to comprehensively address instances of undefined behavior in C++ that cannot be diagnosed at compile time. We are currently working on a paper exploring this approach, [D3100R0], and we are aware of at least one other paper currently in development that addresses integrating Contracts and erroneous behaviour.

However, at present, we do not yet know exactly which extensions to the `std::contracts::contract_violation` API are needed to evolve it into a comprehensive API for runtime detection of undefined behaviour. As such, we consider it premature to add one part of such an API to the Standard, in the form of the single enum value `evaluation_undefined_behavior`, without having had a chance to discuss the other possible parts of such an API, or indeed consider this design space more holistically.

In particular, it is not certain that undefined behaviour detected at runtime should call the same callback as diagnosed contract violations (in particular if this undefined behaviour happens outside of the evaluation of a contract predicate, which is a possibility not yet discussed in any paper targeting SG21 at the time of writing). It seems like a plausible direction, but adding `evaluation_undefined_behaviour` to the API of `std::contracts::contract_violation` now would effectively set this direction into stone before SG21 even had a chance to discuss it, which does not seem compatible with our usual consensus-finding process.

Further, even if runtime detection of undefined behaviour is to re-use the same callback as diagnosed contract violations (that is, the contract-violation handler), it is not certain that this enum value is the correct design choice. If we diagnose instances of undefined behaviour at runtime and report them via this API, we might for example choose to:

— report the defect as an implicit precondition or postcondition predicate that evaluated to `false`, such as an implicit precondition `p != nullptr` on a pointer dereference `*p`, or an implicit postcondition of `false` on a function marked `[[noreturn]]`, in which case the enum value `predicate_false` might make more sense,

— report different kinds of undefined behaviour via different enum values rather than just a single value that captures all of them, for example to programmatically distinguish defects that may compromise memory safety from defects that merely compromise correctness.

Finally, it is not clear how `evaluation_undefined_behavior` would interact with other parts of the `std::contracts::contract_violation` API. Consider again the example of a sanitiser detecting undefined behaviour due to a null pointer dereference. What should `kind()` return in this case?

— If we wish to treat this case as the violation of an implicit precondition `p != nullptr`, we might want to return `contract_kind::pre`.

— Otherwise, we might want to return a non-standard `contract_kind` supplied by the implementation as a vendor extension.

Discussion of this idea on the SG21 reflector revealed that there is no consensus on what the correct choice is.

2.3 Specification challenges

Trying to find wording to correctly specify `evaluation_undefined_behavior` has generated a considerable amount of discussion between the authors of [P2900R4]. At the time of writing, there is no consensus on the correct specification strategy for this feature.

The original wording in [P2811R5] says that the meaning of the enum value is that the "contract predicate would have undefined behaviour when evaluated". This can be misunderstood to mean a

normative requirement, which would be impossible to implement and therefore impossible to specify normatively. The reason this is impossible to implement is that the predicate may call an existing library function or a function otherwise defined in a separate translation unit, and when such a function definition was compiled, it was not compiled with any sort of undefined behaviour detection mechanisms in place. An implementation cannot reasonably detect all undefined behaviour that would occur in the cone of evaluation of a predicate through all possible translation unit boundaries when calling arbitrary functions.

In reality, the specification in [P2811R5] only gives a non-normative encouragement to call the violation handler upon detection of undefined behaviour (with varying degrees of QoI). But even if we do that, it is still an open question how we should specify `evaluation_undefined_behavior` normatively.

One suggested option is to say that no program with well-defined behaviour will ever result in the value `evaluation_undefined_behavior` being passed into the violation handler. This is technically correct, but does not work as a specification, as it makes `evaluation_undefined_behavior` effectively a variant of `std::unreachable()`: the value cannot be meaningfully used by a contract-violation handler in an `if`-statement, because an optimizer will optimise away conditions and branches where this would be the value returned by `detection_mode()`.

Another suggested option is to say that the value `evaluation_undefined_behavior` is passed into the violation handler for contract violations that occurred "for implementation-defined reasons". However, this immediately raises the question what the permissible reasons are, which is a question we currently do not have consensus on (see Section 2.1).

More generally, while we already have some facilities in the Standard that do nothing normatively and exist only for the convenience of tools (`std::breakpoint`, some attributes), we do not yet have a facility in the Standard that only has meaning for programs that have undefined behaviour and are therefore outside of the scope of the Standard itself. We need to figure out how to acknowledge the existence of sanitisers in the Standard; doing so is novel.

One suggestion is that if the behaviour of a program is undefined, it may be permissible for a sanitiser to change the program into a different program that no longer has undefined behaviour, and to thus bring it back into the scope of the Standard where we can specify something about the behaviour of the program. Another suggestion is that it may be possible to modify [defns.undefined] to introduce, for the first time, requirements on C++ programs with undefined behaviour in the Standard. At any rate, we believe it would be better if exploring and reviewing these specification strategies would not block the progression of the Contracts MVP into C++26.

2.4 Lack of usage experience

As we already pointed out, having a sanitiser that not only detects runtime undefined behaviour, but also reports information about where and how it occurred to a user-defined callback, where these properties can be programmatically queried through a standard API, is an idea very much worth exploring. However, as far as we know we do not yet have any usage experience with this approach, raising the question whether adding it to the Standard at this time would be premature.

All clang sanitisers have a callback `__sanitizer_set_death_callback`, but it takes no arguments. The callback can therefore be used to inform the user that the process is about to terminate, but it does not provide any API whatsoever 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*`. However, this merely provides a string with the report dump that will be written to the console, so again this does not provide an API to programmatically query what happened or where. Moreover, this callback is specific to ASan and does not exist for UBSan (see UBSan issue 1298).

In conclusion, at the time of writing we are not aware of any sanitiser or other tool that is capable of detecting undefined behaviour at runtime and reporting meaningful information about the defect to a user callback with an API along the lines of `std::contracts::contract_violation`.

3 `will_continue()`

The class `std::contracts::contract_violation` was added to the Contracts MVP with the member function `will_continue()` at a time when the only possible contract semantics were *ignore* and *enforce*, the latter being the only possible value returned by `semantic()` because *ignore* never results in an invocation of the contract-violation handler.

At the time, `will_continue()` was intended as a futureproofing facility to be able to programmatically distinguish the *enforce* semantic from the not-yet-standard *observe* semantic on an implementation that decides to provide the latter as a vendor extension. Later, the *observe* semantic was added to the Contracts MVP via adopting [P2877R0], making `will_continue()` seemingly redundant (a value of `true` maps to *observe* and a value of `false` maps to *enforce*), but nevertheless `will_continue()` remained in the MVP.

The motivation given for retaining `will_continue()` is usually some combination of the three possible usages discussed below. As we will see, all three have problems.

3.1 Usage for nonstandard checked contract semantics

One possible use case for `will_continue()` in the current MVP is for working with checked contract semantics other than *enforce* and *observe* that an implementation may decide to provide as a vendor extension. It is often not necessary to know the exact behaviour of each such semantic within the contract-violation handler; it is however useful to programmatically distinguish a semantic that continues running the program after the contract-violation handler returns (`will_continue() == true`) from one that will terminate the program. In the latter case, the contract-violation handler can decide to take special measures (such as throwing an exception) to circumvent the termination in applications where it is undesirable.

While this use case is at least somewhat plausible, the problem lies in specifying what it means exactly for the program to "continue" or "terminate" after the contract-violation handler returns and what "after it returns" means; these are currently open questions. Proposal 1.6 in [P2811R5], which was adopted by poll into the Contracts MVP, says that `will_continue()` indicates "whether, when the contract-violation handler returns normally, evaluate is expected to continue immediately following the violated contract-checking annotation". From this specification, it is not clear whether the compiler is allowed (or even required) to report `will_continue() == false` if it can prove that, for example, the next statement after the violated contract assertion is a call to `std::terminate()`:

```
void test(bool b) {
  contract_assert(b);
  std::terminate();  // does it count as "continue" if control flow reaches this line?
}
```

Reasoning about user code that follows the contract assertion seems like an unnecessarily hard task for implementations. The model seems more clear when the property `will_continue()` can be determined solely from the evaluation semantics of the contract assertion itself. And indeed, [P2900R4] describes `will_continue()` along those lines by specifying that it "returns `true` if flow of control will continue into user-provided code should the contract-violation handler return normally, `false` otherwise.

Alas, this specification does not work either. For example, if a vendor-specific checked semantic is specified to call `std::terminate` after the contract-violation handler returns instead of `std::abort` (called by *enforce*), it will call the termination handler, which may run "user-provided code"; if

the semantic is specified to call `std::exit`, then destructors of global objects will be called, which almost certainly will run "user-provided code". The current specification in [P2900R4] suggests that in both cases, `will_continue()` should return `true`, which is counterintuitive and renders the facility useless for this purpose.

There might be a way to specify `will_continue()` such that it would be more useful for reasoning about vendor-specific contract semantics, but at the time of writing it is unclear how this could be accomplished.

3.2 Usage for runtime detection of undefined behaviour

If we are not using `will_continue()` to reason about vendor-specific contract semantics, we are left with using it for the two standard checked semantics, *observe* and *enforce*. At first glance, it seems that `will_continue()` should always be `true` for *observe* and `false` for *enforce*, however [P2900R4] says that for *observe* this "will generally be `true`, but it may be `false` should the platform identify that user-provided code will never execute along the branch where this contract-violation has been detected."

It is not clear from this description in which possible cases the latter would occur, and earlier revisions of [P2900R4] did not give an example. This prompted several confused SG21 members to ask for such an example. The only example that [P2900R4] now provides — and indeed, the only motivation given in the paper for having `will_continue()` in the MVP at all — has to do with runtime detection of undefined behaviour:

```
void f(int* p) {
  contract_assert(p != nullptr);
  *p = 5;
}
```

[P2900R4] argues that if a sanitiser or other tool can prove that all control flow paths which would be reached after the contract assertion is evaluated have undefined behavior (in this case because they dereference a null pointer), it can inform the contract-violation handler of this impending undefined behaviour by returning the value `false` from `will_continue()`.

We believe that motivating `will_continue()` with the desire to use it for runtime detection of undefined behaviour is problematic, because of many of the same reasons why using the value `evaluation_undefined_behavior` for runtime detection of undefined behaviour is problematic (see Section 2). Such usage is inherently unreliable and unportable, hinges on reasoning about code not local to the violated contract assertion, poses specification challenges that we do not yet have agreed-upon solutions for, and suffers from a lack of usage experience. It also offers no way to distinguish between the case where `will_continue()` returns `false` because the implementation detected undefined behaviour following the evaluation of the contract annotation, and the case where `will_continue()` returns `false` because the chosen contract semantic will invoke (well-defined) termination, which feels like an important distinction to make in order to detect the relevant defect.

At the same time, we cannot think of any other case, that is, any program with well-defined behaviour, where it would make sense for `will_continue()` to return `false` for the *observe* semantic.

3.3 Usage as a shorthand for querying the semantic

If we are not using `will_continue()` to reason about vendor-specific contract semantics, and we are not using it for runtime detection of undefined behaviour, then the meaning of `will_continue()` is merely that it is a shorthand for `semantic == std::contracts::contract_semantic::observe`, similar to how `empty()` is a shorthand for `size() == 0`. Such a specification is very clear and does not suffer from any of the problems described earlier in this paper. It also does not seem useful enough to justify the addition of `will_continue()`, and the potential for confusion that comes with it, to the C++ Standard.

4 Summary

In this paper, we have discussed the enum value `evaluation_undefined_behavior` and the function `will_continue()` which are currently part of the standard library API of the Contracts MVP proposal [P2900R4]. We identified several open questions around the meaning, correct specification, and motivating use cases for both facilities.

[P2900R4] motivates both facilities by the desire to programmatically report undefined behaviour detected at runtime by a sanitiser or similar tool. While this is a promising direction that should be explored further, it is not clear whether the proposed API is indeed the most suitable for this task; further, there are currently significant specification challenges as well as a lack of user experience with this approach.

One of these two facilities, `will_continue()`, also has a potential use outside of the context of runtime detection of undefined behaviour. One suggested use case is for reasoning about vendor-specific contract semantics, however it is currently unclear how to specify the function such that it is usable for this purpose. The only other known use case for `will_continue()` is as a shorthand for querying the contract semantic (whether it is *enforce* or *observe*) by calling `semantic()` directly, which does not seem particularly useful.

Overall, we do not believe that a proposal with this amount of open questions can be forwarded to EWG and LEWG for design review. Considering that these open questions are non-trivial; that there is currently a lack of consensus in SG21 on how to resolve them; that it will take time and effort to reach such consensus; and that neither `evaluation_undefined_behavior` nor `will_continue()` are essential in order to use the Contracts MVP, or depended upon by any other part of the Contracts MVP, we propose that both be removed from the Contracts MVP at this stage and instead considered for standardisation as post-MVP extensions, in order to maximise our chances of getting the Contracts MVP into C++26.

References

[D3097R0] Timur Doumler et al. Contracts for C++: Support for virtual functions. Manuscript in preparation, 2024-??-??

[D3098R0] Timur Doumler et al. Contracts for C++: Postcondition captures. Manuscript in preparation, 2024-??-??

[D3100R0] Timur Doumler et al. Contracts, Undefined Behaviour, and Erroneous Behaviour. Manuscript in preparation, 2024-??-??

[P2695R1] Timur Doumler and John Spicer. A proposed plan for Contracts in C++. https://wg21.link/p2695r1, 2023-02-09.

[P2795R4] Thomas Köppe. Erroneous behaviour for uninitialized reads. https://wg21.link/p2795r4, 2023-11-10.

[P2811R5] Joshua Berne. Contract-violation handlers. https://wg21.link/p2811r5, 2023-06-08.

[P2877R0] Joshua Berne and Tom Honermann. Contract Build Modes, Semantics, and Implementation Strategies. https://wg21.link/p2877r0, 2023-06-09.

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

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