Refining Contract Violation Enumerations

Document #:	P3102R0
Date:	2024-02-06
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

The SG21 MVP describes a contract-violation handler that is primarily invoked when a contract violation is detected by observing or enforcing a contract assertion. The replaceable contract-violation handler that users may customize is passed a `contract_violation` object that describes the nature of that violation. This paper explores how the properties of that object, in particular `kind` and `detection mode`, might be used in different scenarios where the contract-violation handler will be invoked and what enumerators those enumerations should contain.

Contents

Revision History

Revision 0

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

With the adoption of [P2811R7], the SG21 MVP containing the consensus proposal for Contracts in C++, [P2900R4], includes full support for a replaceable contract-violation handler. When the contract-violation handler is invoked, the platform will provide an instance of `std::contracts::contract_violation` to that handler. The purpose of this object is to describe, to the violation handler, what happened that resulted in the recognition of a contract violation having occurred.

We have two primary purposes in describing the violation to the handler.

1. Each piece of information that can be reported to the human responsible for the software will help with the prompt diagnosis and remediation of the software defect that led to the contract violation handler being invoked.
2. A violation handler must also make decisions about what forms of automated remediation it might apply — allowing continuation, throwing an exception, termination, or even starting an new event loop inline. Each aspect of how a violation came to be might potentially impact these decisions.

This need for details is balanced with the cost of providing that detail, often based on whether a violation handler could acquire that information separately without the `contract_violation` object being burdened with providing it or the cost of providing it when it might not otherwise be needed. The primary purpose is to encode enough information to be able to identify the immediately local cause for the violation being detected, such that a log message containing that information provides a robust starting point for the diagnostic process.

For any given software behavior that can result in a contract violation being detected and the invocation of the contract-violation handler, the expected properties of the `contract_violation` object must be specified sufficiently to allow contract-violation handlers to reason about what happened. As proposed in [P2811R7], the `kind` and `detection_mode` are intended to be sufficient to allow a violation handler to know two things.

1. The `kind` represents the type of language construct that resulted in the contract violation being detected.
2. The `detection_mode` represents which of the potential possible routes through that language construct resulted in a violation being detected.

1.1 Basic Kinds

The Contacts MVP proposes three forms of contract assertion: precondition assertions, postcondition assertions, and assertion expressions. Distinguishing between these is the `kind` enum's primary purpose.

Proposal 1: Basic Contract Kinds

The `kind` enumeration shall include these enumerators[1]:

- `pre`: Indicate that it was a precondition assertion whose evaluation detected a violation.
- `post`: Indicate that it was a postcondition assertion whose evaluation detected a violation.
- `assert`: Indicate that it was a contract assertion whose evaluation detected a violation.

Note, importantly, that when the `kind` is `pre` or `post`, we are not indicating that it was a precondition or postcondition of the *function* that was violated but rather whether it was a *precondition assertion* or *postcondition assertion* that was violated. One might ask what the difference is, and the answer is subtle. A *precondition* is a requirement that a contract puts on its callers, while a *precondition assertion* combines an algorithm to detect contract violations when a function is invoked. The names are very similar because, largely, these two concepts significantly overlap; in general, if something is wrong as soon as a function is invoked, the error can rarely be the fault of the function being invoked and must be the responsibility of the caller. On the other hand, meaningful and reasonable exceptions to this correlation do exist.

- Class invariants are often checked on function invocation, and though they might be violated due to a caller corrupting the internal state of an object, more often a broken invariant is the result of the library having failed to maintain it after a previous function invocation.
- Some requirements are very expensive to check at function invocation time, yet either they are cheap to check or a heuristic check sometimes identifies problems that can be evaluated when a function is ready to return.
- Some requirements on a caller involve promises not to do something concurrently with the invocation of a function — a promise that can not be validated until the function completes its execution. A postcondition assertion with sufficient supporting infrastructure could certainly validate that such concurrent misuses have not occurred.

1.2 Basic Detection Modes

Each of the contract assertion types in the MVP functions in largely the same way by containing a predicate that identifies contract violations. The MVP also indicates a few discrete routes that can result in a contract violation being detected when a contract assertion is evaluated.

1. The predicate can be evaluated and produce a result of `false`.
2. The predicate can be evaluated and throw an exception.
3. The predicate can be elided in a situation where the compiler can prove that the predicate will always produce a result of `false`.

To capture the kinds and modes of detection that are already specified in the MVP, we thus propose the following enumeration values.

[1] Note that the purpose of this proposal is completeness and clarity; it is not an alteration to the status quo for this enumeration as proposed in [P2900R4].

Proposal 2: Basic Detection Modes

The `detection_mode` shall include these enumerators[2]:

- `predicate_false`: Indicate that a predicate was evaluated and a value of `false` was produced.
- `predicate_would_be_false`: Indicate that a predicate would have evaluated to `false` if it were evaluated.
- `evaluation_exception`: Indicate that some expression was evaluated and an exception escaped the evaluation of that expression.

To minimize the possibility of requiring that new features need to add new enumerators, these enumerators are defined to be as general as they can be yet clearly capture what happened for the cases the MVP distinguishes.

In particular, producing a result of `false` is a property of being a predicate — i.e., an expression that is contextually converted to `bool`. Therefore, both enumerators involving a result of `false` are prefixed with `predicate`. On the other hand, any evaluation that might result in a contract violation being detected, even if that evaluation is not specifically part of a predicate, can result in an exception being uncaught and the contract-violation handler being invoked.

1.3 Future Contract Kinds

The contract-violation handler is intended to be a general purpose central hook for programs to use to handle, in a well-defined way, what would otherwise be undefined behavior. Contract assertions provide a mechanism for configurably transforming library undefined behavior — i.e., precondition violations — into a well-defined and centrally managed invocation of the contract-violation handler.

There are, however, many other possible ways in which we might want to add features to the C++ language that leverage the same hook to manage situations that are otherwise invalid behaviors.

- Preconditions and postconditions can both be seen as special cases of *procedural function interfaces*[3] that wrap a function invocation in a block of code that includes assertions validating various parts of that function's contract. Consider the following two morally equivalent functions, one written using just the contract *kinds* provided by the MVP and another wrapping all checks into a single procedural interface:

```
int f1(int i)
  pre( precondition(i) )
  post( r :  postcondition(i,r));

int f2(int i)
  interface {
    contract_assert( precondition(i) );
    auto r = implementation;
```

[2]This proposal identifies the essential enums in [P2900R4] that capture the modes of detection where a contract violation is mandated, with the only change to the status quo being the addition of the `predicate_would_be_false` enumerator to distinguish when a predicate was not evaluated because its result could be determined without evaluation.

[3]See [P0465R0].

```
  contract-assert( postcondition(i, r);
};
```

While violations of the specific assertions within the interface might use a `kind` of `assert`, an exception escaping the interface must still be treated as a violation of that interface; procedural interfaces would demand a new value for the `kind` enumeration to be used when no more specific value is applicable to a particular contract violation. In this situation in which an exception inappropriately escapes the body of a procedural interface, no new `detection_mode` is needed since the `evaluation_exception` enumerator already completely captures this scenario.

- The existing `assert()` macro provided in `<cassert>` is a contract assertion. The primary differences between an assert macro and a `contract_assert` are relatively small.
  - The assert macro, when disabled, completely elides its predicate's tokens from the translation unit during preprocessing. This aspect of `assert` enables functionality where additional information can be captured in blocks guarded by `NDEBUG` and then used in invocations of `assert`. On the other hand, the predicates in uses of `assert` can quickly become uncompilable if an organization does not regularly build with `NDEBUG` undefined, resulting in an inability to enable `<cassert>` at all, a problem commonly known as *bit rot*.
  - The `assert()` macro, when it detects a contract violation, always terminates the program with a diagnostic message, something that might be inappropriate or insufficient for many organizations.
  - Exceptions escaping from the predicate of the `assert()` macro simply escape.

  Any of these variations could be removed by integrating `<cassert>` more directly with the Contracts facility. For example, having `assert()` invoke the contract-violation handler when it detects a violation would be a very natural extension.

  One could simply dictate that `assert(X)` expands to `contract_assert(X)` when `NDEBUG` is not defined and nothing otherwise, but that introduces a loss of information about what syntactic construct that was actually introduced by a developer ended up identifying a contract violation. This approach would also be a larger breaking change due to the more subtle variations between the evaluation of a contract predicate and the evaluation of `assert()`, such as the handling of escaped exceptions. An improved solution maps `assert()` more directly to its behavior, leaving evaluation untouched (and thus backward compatible with the fewest surprises) and dictating that the contract-violation handler will be invoked. Since this case involves a distinct syntactic construct, distinguishing this case as a different contract `kind` can be important.

  This distinction can greatly help with the migration from platforms where `assert()` has its classic behavior to those where it is integrated with the Contracts facility, making it clear when a new path into the contract-violation handler is the result of (possibly legacy) code that is using `assert()` as opposed to code that is using the Contracts facility directly.

- Sanitizers, such as Address Sanitizer ([asan]) or Undefined Behavior Sanitizer ([ubsan]) provide generally conforming implementations that define certain core-language undefined behavior to crash the program with diagnostics. These tools have historically been designed as diagnostic

tools, not for production use; therefore, they do not provide features key to deployment, such as a customizable violation handler that gets provided with diagnostic information.

The sanitizers currently do provide a somewhat limited mechanism to hook into a user-provided callback when a violation is detected, with a number of APIs of varying levels of functionality using `__sanitizer_set_death_callback` or a sanitizer-specific variation of that function, such as `__msan_set_death_callback` or `__asan_set_error_report_callback`. Changing these functionalities to invoke a contract-violation handler would allow for more reasonable deployment of sanitized builds into production environments without having to reinvent logging and reporting mechanisms for each particular sanitizer.

Since no explicit syntactic construct triggered a particular sanitizer to detect a bug, having sanitizers use a new `kind` to report their violation is the natural solution. The other properties of the contract violation are not as obvious. For example, a sanitizer might not know the source location where a bug originates, — but providing an empty value for the `location` property is certainly acceptable. More importantly, none of the values for `detection_mode` proposed above make sense for what triggered a sanitizer to detect a violation. What is currently happening with sanitizers is that something that is about to happen will have undefined behavior, and the sanitizer preempts that behavior by instead printing a diagnostic and terminating the program. The enumerators mentioned above for `detection_mode` do not fit this scenario; there is no predicate the user provided whose evaluation is or would be `false` nor is there an exception being thrown from any evaluation. Instead, the contract violation should capture that it is being called because some evaluation would have had undefined behavior; therefore, the correct value for `detection_mode` that sanitizers should pass is `evaluation_undefined_behavior`.

- The C Standard's Annex K[4] provides a mechanism to set a handler for various precondition checks on safer versions of many aspects of the C language API. When a runtime constraint is detected, a constraint violation handler is invoked (which can be set at runtime with the `set_constraint_handler_s` function).

  The behavior of the default runtime constraint handler is implementation defined, which certainly leaves room for implementations to do the natural thing and have that default handler invoke the contract-violation handler. In such a case, considering that the runtime constraints in Annex K are generally expressed as preconditions on inputs, the `contract_violation` object could be populated with a `kind` of `pre`, but in general and to have a direct path for a developer to take a logged contract violation and use it to begin to identify and fix the detected problem, indicating that the kind was an Annex K runtime constraint violation would be much more useful. Therefore, providing a distinct `kind` value to start diagnosis on the right path would generally be better. Most preconditions in Annex K are expressed as conditions applied to inputs, so a detection mode of `predicate_false` with a `comment` containing the condition would generally be the most consistent way to express such violations.

- Much like what happens when building with a sanitizer, any evaluation that has undefined behavior could conceivably invoke the contract-violation handler, largely because that evaluation could conceivably do anything at all due to the C++ Standard's complete lack of constraints on a program with undefined behavior.

[4]See [isoc11], "Annex K," p. 582.

In fact, many core-language undefined behaviors could be detected with a locally introduced check and would not require the level of instrumentation that most of the sanitizers do[5]:

- Arithmetic operations on signed integral types that overflow
- Dereferencing a *null* pointer
- Falling off the end of a function with a non-`void` return type
- Returning from a function having the `[[noreturn]]` attribute[6]
- Out of bounds indexing into an array of known bounds
- Reading or writing from the padding bytes of an object
- A `static_cast` downcasting a polymorphic type where the concrete type is not correct, i.e., where the corresponding `dynamic_cast` would have returned `nullptr`
- `reinterpret_cast` of pointers that violates the alignment requirements of the target type
- Division by zero or modulus by 0
- Bit shifting by a negative number of bits or by a number of bits greater than the width of the left operand.
- Invoking a pure abstract function, particularly during construction or destruction of an object
- Invoking unsequenced operations where one has a side effect on the same memory location used in the other, such as `i++ + i`. Note that cases where the same variable is used could statically become violations, while cases where this UB results from aliases could be checked by inserting checks on the addresses being modified and accessed.

The interesting case arises when a compiler chooses to use the freedom of a behavior being undefined to take one of the above situations and add appropriate checks that fully define the behavior and result in invoking the contract-violation handler when there would otherwise have been language-level undefined behavior.

Each of the above could conceivably be considered a violation of a precondition imposed by the language itself. The `kind` could conceivably be `pre` here, but that would misdirect diagnosticians into looking for a `pre` specifier on a function that is not actually there. Instead, a new `kind` to indicate that language undefined behavior was being checked would be appropriate. Again, as with the sanitizers, there is not, a priori, a predicate that evaluated to `false` when there is a violation, so in general, using a different enumerator for `detection_mode` would make more sense. In particular, the detection mode of `evaluation_undefined_behavior` again captures the situation that was identified completely and that would be the appropriate value for a platform to use.

[5] This list is not meant to be exhaustive.

[6] Note, however, that attributes which introduce undefined behavior, such as `[[noreturn]]` and `[[assume]]`, might instead benefit from easier identification through a new value for `kind` since the contract violation is tied directly to a piece of syntax denoting an erroneous situation. A similar approch could be taken for future such attributes, such as `[[throws_nothing]]` as proposed in [P2946R1].

Considered together, the above situations consistently indicate that an extension value for `kind` is appropriate since each new potential source of contract violations is added by a platform. Doing this greatly improves the ability to quickly map a reported contract violation to an understanding of what steps led to the situation.

What is consistently needed, however, is a vocabulary to cover the cases that involve contract violations but do not involve a user's predicate being evaluated or an exception being thrown inappropriately. In these cases, an enumerator to indicate that something was going to happen that would be undefined behavior is needed. Therefore, we propose that enumerator be included as well.

Proposal 3: Undefined Behavior Detection Modes

The `detection_mode` shall include this enumerator[7]:

- `evaluation_undefined_behavior`: Indicate that an evaluation would have had undefined behavior if it had continued.

2 Conclusion

Having a common vocabulary for not only the already mandated mechanisms for invoking a contract-violation handler, but also for understanding various potential extensions allows users to write robust contract-violation handlers and then leverage them for many years to come across an increasingly wide variety of platforms.

Maximizing what we can distinguish when contract violations occur maximizes the utility of Contracts in general, largely by minimizing the cognitive steps needed to go from a reported contract violation to an understanding of what actually went wrong in a program.

All put together, this results in a clear set of useful enumerators to include in the `kind` and `detection_mode` enumerations:

```
namespace std::contracts {
enum class contract_kind : int {
  pre,     // A precondition assertion identified a violation.
  post,    // A postcondition assertion identified a violation.
  assert   // An assertion expression identified a violation.
};
}

namespace std::contracts {
enum class detection_mode : int {
  predicate_false,               // A predicate evaluated to false.
  predicate_would_be_false,      // A predicate would evaluate to false if evaluated.
  evaluation_exception,          // Something was evaluated, and an exception escaped.
  evaluation_undefined_behavior  // Something would have undefined behavior if evaluated.
};
}
```

[7] This enumerator is already present in [P2811R7] and [P2900R4], and we hope this paper has clarified its intent.

Acknowledgements

Thanks to Lisa Lippincott for pointing out the lack of a method to identify when the evaluation of a contract assertion's predicate was elided. Thanks to Timur Doumler for questioning the clarity of the purpose of the `evaluation_undefined_behavior` enumerator.

Bibliography

[asan] Google, "Address Sanitizer (ASan)"
https://clang.llvm.org/docs/AddressSanitizer.html

[ubsan] Google, "Undefine Behavior Sanitizer (UBSan)"
https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[isoc11] *ISO/IEC 9899:2011 Information Technology — Programming Languages — C* (Geneva, Switzerland: International Standards Organization, 2011)
https://www.iso.org/standard/57853.html

[P0465R0] Lisa Lippincott, "Procedural Function Interfaces", 2016
http://wg21.link/P0465R0

[P2811R7] Joshua Berne, "Contract-Violation Handlers", 2023
http://wg21.link/P2811R7

[P2900R4] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R4

[P2946R1] Pablo Halpern, "A flexible solution to the problems of `noexcept`", 2024
http://wg21.link/P2946R1