Fixing the library API for contract violation handling

Gašper Ažman (gasper.azman@gmail.com)
Timur Doumler (papers@timur.audio)

Document #: P3227R1
Date: 2024-10-24
Project: Programming Language C++
Audience: SG21, LEWG

Abstract

This paper proposes several improvements to the contract-violation handling API of the Contracts MVP. One change enables a user-defined contract violation handler to intercept termination of the program by the contract evaluation semantic – which is intended to be possible, but is not *actually* possible with the current API. Another change makes it easier and less error-prone to query whether an exception was thrown during evaluation of a contract predicate and to retrieve that exception. A third change improves the usability of the provided enums.

1 Motivation

The current Contracts MVP proposal [P2900R9] provides the ability to install, at link time, a user-defined contract-violation handler. Information about the contract violation and the contract evaluation semantic is passed to this handler in the form of an object of type `contract_violation`.

In particular, the handler might want to branch on the following two boolean queries:

1. Will the contract-violation handling mechanism attempt to terminate the program after the handler has returned? If the latter is the case, we may want to prevent this termination from happening by throwing an exception from the handler;
2. Did the predicate evaluate to `false`, or did its evaluation exit via an exception? If the latter is the case, we may want to retrieve that exception, and handle or rethrow it.

Unfortunately, the library API for contract violation handling currently proposed in [P2900R9] does not provide adequate tools for making the above queries.

2 Adding the necessary member functions

2.1 Detecting a terminating semantic

The currently proposed API for detecting a contract evaluation semantic under which the contract-violation handling mechanism will attempt to terminate the program after the handler has returned – consists of a member function `semantic()` which returns an enum `evaluation_semantic`, defined as follows:

```
enum class evaluation_semantic : unspecified {
  enforce = 1,
  observe = 2,
  // additional implementation-defined enumerators
};
```

If, in the contract-violation handler, we know that no implementation-defined additional semantics are present, then (and only then) can this enum answer the question of whether we are dealing with an evaluation semantic that will attempt termination:

```
void handle_contract_violation(const& contract_violation) {
  if (violation.semantic() == evaluation_semantic::enforce)
    throw DoNotTerminate();
}
```

However, if the implementation provides *any* additional implementation-defined semantics, we cannot portably determine whether the semantic will attempt termination or not, since there is no API available to query whether that is the case. In other words, we do not have a way to ask this question unless we can write an exhaustive list of all available evaluation semantics, which is inherently non-portable.

Furthermore, the full set of semantics may only be known at link-time. Imagine a sanitizer (perhaps ThreadSanitizer) that emits a separate semantic for some kind of violation (say a detected race condition, which may or may not be benign), and only one translation unit is built with it. Without knowledge of such tooling at handler design time, even the Standard Library cannot provide such a query if given merely an `evaluation_semantic` value.

Instead of switching on concrete semantics, what the handler needs is a boolean query "*will the current evaluation semantic attempt to terminate the program*"? We call such a semantic a *terminating semantic*. We can easily enable this query by adding a new member function `is_terminating` that returns `bool`. With this addition, the above code becomes:

```
void handle_contract_violation(const& contract_violation) {
  if (violation.is_terminating())
    throw DoNotTerminate();
}
```

which expresses the user's intent unambiguously and succinctly.

2.2 Handling a predicate evaluation exception

In the current API, determining whether predicate evaluation exited via an exception, and retrieving that particular exception, requires the following incantation:

```
void handle_contract_violation (contract_violation& violation) {
  if (violation.detection_mode() == detection_mode::evaluation_exception)
    my::handle(std::current_exception());
}
```

Even when omitting `std::contracts::` qualifiers (as we do throughout this paper), the above is rather cryptic. An average C++ developer might not be able to tell what is actually going on in the code above. Even worse, they might be tempted to simply query `std::current_exception()` to retrieve the exception:

```
void handle_contract_violation (contract_violation& violation) {
  if (auto ex = std::current_exception())
    my::handle(ex);
}
```

However, this is incorrect, as `std::current_exception()` may also point to an unrelated exception that was in the process of being handled when the contract violation occurred:

```
try {
  my_vector.at(idx);  // throws std::out_of_range exception
}
catch (std::out_of_range) {
  contract_assert(database_valid());  // evaluates to false, calls handler
  // code handling std::out_of_range exception...
}

void handle_contract_violation (contract_violation& violation) {
  if (auto ex = std::current_exception())
    // `ex` now points to the earlier std::out_of_range exception :(
}
```

This double duty of `std::current_exception` does not lead to a good user experience. We need a dedicated API to retrieve specifically the exception thrown during the contract check. This is easy to achieve by adding a new member function `evaluation_exception` that returns a `std::exception_ptr` to that exception if there is one, or an empty `std::exception_ptr` otherwise. This eliminates the need to use the `detection_mode` enum and the function `std::current_exception` in the contract-violation handler at all:

```
void handle_contract_violation (contract_violation& violation) {
  if (auto ex = violation.evaluation_exception())
    my::handle(ex);
}
```

3 Discussion

3.1 Previous attempts at detecting termination

Earlier versions of the Contracts MVP had the member function `will_continue()` which was adopted via [P2811R7] and was intended to solve the same problem as our proposed `is_terminating()`.

The initially proposed specification of `will_continue()` was that it should return `true` if "evaluation will continue after the violated contract check should the contract-violation handler return normally." This somewhat vague specification (what does "continue" mean, exactly?) was made more precise when [P2811R7] was merged into the Contracts MVP: it then said that `will_continue()` should return `true` if "flow of control will continue into user-provided code should the contract-violation handler return normally".

However, it turned out that this is not only very difficult to specify correctly, but also the wrong question to ask. If a contract assertion is evaluated with a semantic that will attempt termination, such as *enforce*, the exact mode of termination is implementation-defined and may result in destructor calls and/or calls to cleanup functions such as `std::atexit`, `std::terminate_handler`, etc. which all qualify as user-defined code. Therefore, `will_continue()` may return `true` even if the program will actually terminate.

It was unclear at the time how to fix this broken specification. Therefore, `will_continue()` was removed from the Contracts MVP via [P3073R0]. Having grown wiser, we now know that the right question to ask in the contract-violation handler is not "will evaluation continue?" or "will evaluation continue into user-defined code?", but "will the contract-violation handling mechanism attempt to terminate the program?" which is exactly how our proposed new member function `is_terminating` is specified. This is also the reason why the proposed name starts with "is", not with "will" – we are querying a property that expresses intent, not predicting the future.

3.2 Enforcing semantic vs. terminating semantic

[P2900R9] currently has the notion of *enforcing semantics*, and specifies that *enforce* and *quick_enforce* are both enforcing semantics. No normative definition for "enforcing semantics" is included in the wording, other than that *enforce* and *quick_enforce* are both enforcing semantics, but the front matter states that enforcing semantics "do not allow program execution to continue past an identified contract violation". It has therefore been suggested that a better name for our proposed new member function would be `is_enforcing`, or perhaps `is_enforced` (since the violated contract assertion is being enforced) and that this function should be specified to return `true` if the violated contract assertion was evaluated with an enforcing semantic.

However, it is important to note that "does not allow program execution to continue past an identified contract violation" is not synonymous with "will attempt to terminate the program on contract violation", which is the property that our newly proposed member function is

querying. For example, one could imagine an evaluation semantic that enforces a contract assertion – does not allow program execution to continue past an identified contract violation – by throwing an exception,[1] or perhaps by stalling the affected thread indefinitely. Such semantics would be enforcing in the sense of [P2900R9], but would not attempt to terminate the program on contract violation, and we would therefore want to treat them differently from *terminating semantics*. In particular, we would only want to throw our own exception from the contract-violation handler to prevent termination if we are dealing with a terminating semantic, not just any enforcing semantic. We therefore need to carefully distinguish these cases.

In this paper, we are specifically addressing the need to identify a terminating semantic. To avoid further confusion, we propose that both the front matter and the proposed wording in [P2900R9] be changed to replace the term *enforcing semantics* with *terminating semantics*, to normatively define the latter term, and to classify *enforce* and *quick_enforce* as terminating semantics.

If, in the future, compiler vendors or the C++ Standard itself will introduce enforcing semantics that are not terminating semantics, we can then reintroduce the former term and introduce another member function querying the associated property. We are not proposing to do so now. We should introduce such queries as the need for them arises; we cannot possibly predict the full set of properties that one might want to query in the contract-violation handler in the future, and we should generally avoid adding entities to the Standard unless there is a clear use case and we are sure that we are not cutting off design space.

3.3 Rephrasing semantics as properties

One way of thinking about contract evaluation semantics more generally is to consider them combinations of a set of orthogonal properties. This idea has been explored more thoroughly in [P3237R0]. The current four semantics in [P2900R9] can be represented in a matrix as follows (note that not all possible combinations of properties make sense):

	determines the value of the predicate?	calls the contract-violation handler?	terminates the program on violation?
ignore	no	no	no
observe	yes	yes	no
enforce	yes	yes	yes
quick_enforce	yes	no	yes

We can define more properties in addition to the three listed above. Such additional properties are currently not needed to distinguish the four evaluation semantics in [P2900R9], but may become useful for distinguishing semantics if compiler vendors or the C++ Standard itself start introducing more of them. In the previous paragraph, we have seen

[1] Note that this is not entirely contrived as such a contract evaluation semantic – called "*Eval_and_throw* mode" at the time – was in fact proposed in [P2698R0].

one such property, "allows program execution to continue past an identified contract violation?" – the *enforcing* property – which is distinct from the *terminating* property. As shown in [P3237R0], this matrix can be expanded with even more properties, such as "is the predicate assumed to be `true` after the assertion?" or "what is the mode of termination?" to accommodate possible future semantics, such as the *assume* semantic or the *terminate* semantic proposed in [P3205R0].

It seems tempting to remove the notion of "evaluation semantics" altogether and instead describe the possible behaviours when evaluating a contract assertion with the above properties, or at least make the properties the primary entities and demote the named semantics to aliases for certain combinations.

At first glance, it seems that talking about properties rather than semantics is more helpful, as this allows branching directly on the relevant property rather than having to switch on the entire `evaluation_semantic` enum – whose values will never be exhaustively known – for every such property. In this paper, we propose to map one such property, whether the current semantic will attempt termination – the only property that, with the current set of four semantics, is not already known when the contract-violation handler has been called – to a function querying that property directly in that handler.

However, upon deeper contemplation, focusing on properties as the primary entities seems unwise. Defining the possible semantics via orthogonal properties was existing practice in the C++2a Contracts proposal [P0542R5], which had a *contract level* (default, audit, axiom), a *build level* (off, default, audit), a *continuation mode* (on, off), and at some point a proposed *assumption mode* (on, off) (see [P1710R0], [P1711R0], [P1730R0]). Reasoning about which contract evaluation semantics are actually possible, and which of these are desirable, when combining these properties turned out to be very difficult and confusing (see [P1421R0]). The introduction of named semantics as the primary entities in the Contracts MVP has proven to be helpful as it allows users to compartmentalise the possible behaviours of evaluating a contract assertion in a way that they would not do otherwise.

Therefore, while we do believe that adding functions that let the user query such properties or traits of the evaluation semantics in the contract violation handler is useful, and we propose one such function here, we do *not* propose to replace named semantics with properties as the primary entities as proposed in [P3237R0].

3.4 What to do with the enumerations

In order to query whether the current semantic is a terminating semantic, we deliberately propose a member function `is_terminating()`, which can only be called inside the handler, instead of a free function `is_terminating_semantic(evaluation_semantic)`. As we concluded in Section 2.1, the Standard library cannot, in general, provide such a free function, as the full set of available semantics may only be known at link-time. At this point, the `evaluation_semantic` enum is no longer useful for querying this property.

Similarly, with the addition of a new member function `evaluation_exception`, the `detection_mode` enum is no longer useful for querying whether an exception was thrown during the contract check, and retrieving that exception.

Finally, extensible enums are generally problematic because it is impossible to write an exhaustive switch; worse, a user might write a switch that they think is exhaustive, and when the C++ Standard or a compiler vendor adds a new semantic or detection mode later, they break that user's code.

However, it turns out that the primary use case for either enum is not branching or switching at all, but *logging*, which is very helpful in a contract-violation handler. With the enum, logging the current semantic or detection mode – including any vendor-specific options – can be accomplished in a single statement, rather than having to branch on multiple properties for this purpose.

A secondary use case for the `evaluation_semantic` enum is to serve as a vocabulary type to refer to evaluation semantics in other contexts. No such context besides the contract-violation handler currently exists in [P2900R9]. However, compiler vendors may provide, as an extension, labels on the assertion to specify the desired semantic explicitly (we know of at least one vendor who plans to provide this). The enum is very helpful for this purpose. It seems preferable to not have to switch to a different enum if such labels are upgraded from vendor extensions to Standard features in a later standardisation cycle (see [P2755R1] for a discussion of such labels).

We therefore do *not* propose to remove either enum. However, we do propose to make it clear (via a Note in the specification) that the `evaluation_semantic` enum is intended for logging and as a vocabulary type, and not for querying whether the current semantic is a terminating semantic. To make the `evaluation_semantic` enum more useful as a vocabulary type for labels, we further propose to add enumeration values for *all four* possible semantics, not just the two that may call the contract-violation handler.

[P3237R0] proposes to change `evaluation_semantic` from an `enum` to a `struct` with four fields: `checks`, `calls_handler`, `assumed_after`, and `terminates`, which maps to four distinct properties of possible evaluation semantics. Alternatively, `evaluation_semantic` could remain an `enum` but the numerical values could be chosen such that the bits of that value directly map to the distinct properties, rather than just enumerating the possible semantics with `1`, `2`, `3`, and `4`.[2]

However, in this paper we deliberately do *not* propose to associate the numerical values of this enum with any particular properties of the semantic, such as whether it is a terminating semantic, or to give any further meaning to those numerical values at all. Such properties should be queried via functions like the proposed `is_terminating()`, not via the value of `evaluation_semantic`.

[2] It has been suggested that it would be strange if one of these enums, if zero-initialised, would not have a valid value, and that therefore the enum value for `ignore` should be `0`. However, we do not propose such a change here as we believe it is useful to be able to detect the case where the enum has not been explicitly initialised with a valid value.

We do not want to prescribe whether or how such properties should be represented in the ABI; we do not want to preclude implementations where whether `is_terminating()` returns `true` is potentially dependent on runtime configuration that is not directly represented in the value of the semantic; finally, we cannot conclude at this point that we have truly exhaustively explored this entire space and will never come across contract evaluation semantics that do not have those properties, or where those properties do not make sense.

4 Proposed wording

We propose the following changes to [P2900R9].

- Modify [basic.contract.eval] as follows:

  A contract assertion may be evaluated using one of the following four *evaluation semantics*: *ignore*, *observe*, *enforce*, or *quick_enforce*. ~~The ignore semantic is a *non-checking semantic*; observe, enforce, and quick_enforce are *checking semantics*; enforce and quick_enforce are *enforcing semantics*.~~
  A *checking semantic* is an evaluation semantic which determines the value of the predicate to detect a contract violation; a *terminating semantic* is an evaluation semantic which will prevent program execution from continuing past a violated contract assertion by terminating the program. Observe, enforce, and quick_enforce are checking semantics; enforce and quick_enforce are terminating semantics.

- Add the following new member functions to to class `contract_violation`:

  ```
  bool is_terminating() const noexcept;
  ```

  Returns: `true` if the current evaluation semantic is a terminating semantic, i.e., if the contract-violation handling mechanism will attempt to terminate the program after the contract-violation handler has returned; `false` otherwise.

  ```
  exception_ptr evaluation_exception() const noexcept;
  ```

  Returns: If the contract violation occurred because the evaluation of the predicate exited via an exception, an `exception_ptr` that holds either a copy or a reference to that exception object; otherwise, an empty `exception_ptr`.

- Modify the enumeration `evaluation_semantic` as follows:

```
enum class evaluation_semantic : unspecified {
  ignore = 1,
  ~~enforce = 1,~~
  observe = 2,
  enforce = 3,
  quick_enforce = 4
};
```

[*Note:* ~~No enumeration values for the ignore or quick_enforce semantics are provided because evaluations with those evaluation semantics cannot result in a call to the contract_violation handler.~~This enumeration is intended for logging and as a vocabulary type. To determine whether the current evaluation semantic is a terminating semantic, `is_terminating()` should be used instead. — *end note*]

Revision history

R0 → **R1:** Fixed minor wording bug (green highlighting was in the wrong place)

Acknowledgements

Many thanks to Eric Fiselier, Andrei Zissu, Joshua Berne, and Lisa Lippincott for discussing and reviewing this proposal and providing valuable feedback.

References

[P0542R5] Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. "Support for contract based programming in C++". 2018-06-08

[P1421R0] Andrzej Krzemieński. "Assigning semantics to different Contract Checking Statements". 2019-01-18.

[P1710R0] Ville Voutilainen. "Adding a global contract assumption mode". 2019-06-17

[P1711R0] Bjarne Stroustrup. "What to do about contracts?". 2019-06-13

[P1730R0] Hyman Rosen, John Lakos, and Alisdair Meredith. "Adding a global contract assumption mode". 2019-06-14

[P2698R0] Bjarne Stroustrip. "Unconditional termination is a serious problem". 2022-11-18

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos: "A Bold Plan for a Complete Contracts Facility". 2024-04-11

[P2811R7] Joshua Berne: "Contract-violation handlers". 2023-06-27

[P2900R9] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2024-10-11

[P3073R0] Timur Doumler and Ville Voutilainen: "Remove evaluation_undefined_behavior and will_continue from the Contracts MVP". 2024-01-27

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

[P3237R0] Andrei Zissu: "Matrix Representation of Contract Semantics". 2024-04-15