Nonthrowing Evaluation Semantics

Document #:	P4298R0
Date:	2026-07-15
Project:	Programming Language C++
Audience:	EWG

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

Abstract

C++26 Contracts support having a contract-violation handler that might throw an exception for the purposes of attempting to recover. This essential behavior is needed in many domains, but introduces concerns in others where no exception could escape safely or without the need for significant additional overhead. To handle deciding locally whether to allow exceptions on contract violations we introduce two new contract-evaluation semantics, *'noexcept'-enforce* and *'noexcept'-observe*, which do exactly what their names imply — act just like *enforce* and *observe* but terminate if the contract-violation handler throws an exception.

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

Undefined behavior can unquestionably throw an exception — it can do anything. A contract violation, which often guards against entering undefined behavior, can thus naturally be expected to potentially throw an exception as well. This, however, raises concerns in a variety of ways:

- When an exception emanates from undefined behavior there is no onus on the compiler and runtime to make that exception propagate properly. Objects on the stack might not be destroyed, execution might jump to unexpected places, and there are no guarantees about the soundness of trying to leverage an exception when such bugs are encountered. If we instead have the well-defined (yet erroneous) behavior associated with a contract violation emit an exception the compiler and the program are more obliged to deliver fully defined behavior as the exception unwinds — control must flow to the correct handlers, automatic variables must all be destroyed, etc. To make sure this can happen additional instructions must be emitted and, possibly, many optimizations must be sacrificed.
- Outside what the compiler must guarantee, many library developers feel they must write exception-safe code even when their own code has bugs that would trigger a contract violation. Such developers might be well served by being able to simply prevent any exception from propagating through their call stacks when they violate a contract assertion.
- With the potential adoption of [P3400R4], assertion-control objects (i.e., labels) could allow for selecting locally whether a given contract assertion should support exceptions escaping from a violation of a particular assertion. This could be done with a new facet, or it could be done by introducing new semantics that would allow users to express this desire both in code and on the command line when building their libraries and applications.
- The library API for invoking the contract-violation handler, introduced by [P3290R6], already includes support for `noexcept` entry points into the contract-violation handling process.

To that end, the right solution seems to be to capture the permission to throw an exception as part of the contract-evaluation semantic by introducing variants of those evaluation semantics that invoke the contract-violation handler (*observe* and *enforce*) that do so in a `noexcept` context that will terminate when an exception escapes the contract-violation handler.

This maximizes the opportunity for control without removing anyone's ability to use throwing violation handlers in the situations where they do have a need to, or gain benefits from, explicitly disallowing it.

2 Proposal

Introducing new contract-evaluation semantics only needs to touch on a small number of places:

- **New Semantics**: Add two new contract-evaluation semantics, *'noexcept'-enforce* and *'noexcept'-observe*. These semantics, when chosen, will determine the result of the predicate and trigger the contract-violation handling process if the result is not `true` (just like *enforce* and *observe*). Unlike *enforce* and *observe*, if the resulting call to the contract-violation handler exits with an exception, `std::terminate()` will be invoked.

  Note that, by using `std::terminate()`, we allow implementations to achieve this effect by either wrapping a call to the violation handler in a `try ... catch` that invokes `std::terminate()` *or* by using a separate function for that dispatch which is marked `noexcept`. In practice, prototype implementations all use a separate function for this purpose as it minimizes the code needed for the assertion itself.

- **New Enumerators**: With new semantics come new enumerators `noexcept_observe` and `noexcept_enforce` in `std::contracts::evaluation_semantic` to represent the new semantics.

Then, if [P3290R6] is adopted first, update its behavior to pass through the new semantics when using the `noexcept` entry points:

- **Library API Behavior Changes**: The library API for contracts, introduced in [P3290R6], introduces `nothrow_t` overloads of the functions `handle_enforced_contract_violation` and `handle_observed_contract_violation`. These should be updated to report the enumerators for *'noexcept'-enforce* and *'noexcept'-observe* respectively in the `contract_violation` objects they populate and pass to the contract-violation handler.

Finally, if [P3400R4] is adopted, add a new classifier to the set of evaluation semantic classification functions introduced by that paper:

- **Library API Addition**: With a new classification for contract evaluation semantics, nonthrowing, add a new characterization static function to the set of functions for characterizing evaluation semantics in `std::contracts` that are introduced by [P3400R4]:

  ```
  constexpr bool is_nonthrowing(evaluation_semantic s);
  ```

3 Additional Rationale

Certain fine points in this proposal have additional reasoning for why they are presented as is instead of possible alternatives.

- **Type of Termination**: Unlike when a violation handler throws from the `assert` macro (as proposed in [P3290R6]) we propose that an exception from the contract-violation handler where one of the new semantics has been used results in invoking `std::terminate()`.

  There are, potentially, at least three options we could consider:

  1. Leave implementations additional freedom and just require that the program be contract-terminated when an exception escapes. This would allow implementations to choose between `std::abort()`, `std::terminate()`, or some other form of trap or program termination.

  2. Invoke `std::abort()` the way that [P3290R6] proposes for `assert`. Unlike `assert`, which is an existing contract-checking facility that has a known method of termination, a contract

assertion has no such precedent to follow and committing to `std::abort` does not seem warranted.

3. Invoke `std::terminate()`, the way that any other violation of something the language uses `noexcept` to provide would choose to end program execution.

Because we have chosen to name the new semantics with the use of `noexcept` as an adverb, the choice with the least surprise for users is to invoke `std::terminate()`. It is also the most straightforward solution to implement, as code generated for contract assertions with this semantic just needs to call into an entry point that is marked `noexcept`.

- **The use of semantics**: Potential ideas of providing control over exceptions escaping from the default violation handler using a new facet of labels (as specified in [P3400R4]) were explored, but those miss the point that the decision to allow exceptions to escape is often not one that is made locally within a library but rather one that might be guided by performance requirements in certain environments, or when a builder of a program decides that they never intend to deploy a throwing violation handler. Because of that, encoding the decision in the evaluation semantic extends the contracts facility with maximal flexibility.
- **Naming of the semantics**: Just like *quick-enforce*, we chose to designate the new semantics with a prefix (making `noexcept` grammatically a mid-position adverb) before the base semantic (that is always a verb whose object is the assertion). We see no reason to deviate from that precedent.

4 Implementation Experience

A variant of this proposal was originally suggested and implemented by Ville Voutilainen. It has since been implemented in this form in the P3850 branches of both GCC and Clang that are available on Compiler Explorer, including the interactions with the other contracts extensions those branches carry. A live example that pairs each evaluation semantic with a throwing and a nonthrowing violation handler is available on Compiler Explorer: `https://godbolt.org/z/aWhEzq6Wb`. Both implementations make these features available with the use of the `-fcontracts-p4298` command-line flag. They are also available with the flag `-fcontracts-p3850` that enables prototype implementations of many of the papers described in the overall plan in [P3850R1].

The two new semantics were added as ordinary `evaluation_semantic` enumerators, which let them flow through the entire existing resolution pipeline — the `-fcontract-evaluation-semantic=` default, per-group configuration, JSON configuration files, and the assertion-control labels of [P3400R4] (both the `allowed_semantics` and `compute_semantic` facets) — with no new resolution machinery, only a wider set of possible results. Selecting *'noexcept'-enforce* or *'noexcept'-observe* through any of those channels behaves as expected.

The termination behavior itself required very little new work. The contract-violation-handling ABI shared by the two branches already exposes `noexcept` variants of its handler-dispatch entry points — the same mechanism the `noexcept` entry points of [P3290R6] rely on — so implementing the new semantics amounted to routing their handler call through that existing noexcept boundary rather than the ordinary one. A handler that returns normally behaves exactly as it does under *enforce* or *observe*; only an exception escaping the handler is treated differently, by calling `std::terminate()`.

The `nothrow_t` overloads of the library API introduced by [P3290R6] were updated to report the new semantics, but this raised a question beyond the scope of what the C++ Standard itself specifies — whether this change in behavior should be based on the feature being available in the *calling* translation unit or baked into the runtime library. (The Standard itself, of course, only talks about programs where everything in the program conforms to the same version of the C++ Standard.) We chose to follow the flag of the invoking translation unit since this is currently an experimental feature and programs that never enable the new feature should not be exposed to the new semantic enumerators. Achieving this without an ODR violation required a small novelty: the gated overloads are placed in an inline namespace selected by the feature-test macro, so that the two behaviors are distinct library symbols rather than one symbol with two possible definitions. This detail is an implementation technique, not something the wording need mandate, but it demonstrates that the per-caller behavior is realizable.

During constant evaluation, where no exception can escape a contract-violation handler in the first place, *'noexcept'-enforce* and *'noexcept'-observe* are indistinguishable from *enforce* and *observe* respectively, and were implemented to behave identically to them.

The implementations predefine a feature-test macro, `__cpp_contracts_nonthrowing_semantics`, when the semantics are enabled (the prototype uses the implementation date as its value); the wording proposes it with the customary placeholder value.

One incidental observation is worth noting: the numeric values of enumerators in this paper will, of course, depend on the order in which this and other papers get adopted. For example, [P3100R7] proposes another new evaluation semantic, *assume*, and brings with it its own new enumerator. This is of course an issue for any experimental paper and implementation, and will be a non-issue if and when the papers with new enumerators are adopted into the C++ draft.

5 Wording Changes

These wording changes are relative to the C++29 working draft with git hash 14ed707b, last modified on Wed, 17 Jun 2026 14:18:34 +0100.

6 **Basics** [basic]

6.11 **Contract assertions** [basic.contract]

6.11.2 **Evaluation** [basic.contract.eval]

Modify section 6.11.2[basic.contract.eval], paragraphs 1-2:

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

2 It is implementation-defined which evaluation semantic is used for any given evaluation of a contract assertion.

[*Note 1*: The range and flexibility of available choices of evaluation semantics depends on the implementation and need not allow all ~~four~~six evaluation semantics as possibilities. The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

Modify section 6.11.2[basic.contract.eval], paragraphs 8-13:

8 If a contract violation occurs in a context that is manifestly constant-evaluated (7.7.7[expr.const.defns]), and the evaluation semantic is a terminating semantic, the program is ill-formed.

[*Note 2*: A diagnostic is produced if the evaluation semantic is observe (4.1[intro.compliance]) or `noexcept-observe`. — *end note*]

[*Note 3*: Different evaluation semantics chosen for the same contract assertion in different translation units can result in violations of the one-definition rule (6.3[basic.def.odr]) when a contract assertion has side effects that alter the value produced by a constant expression.

[*Example 1*:

```
constexpr int f(int i)
{
  contract_assert((++const_cast<int&>(i), true));
  return i;
}
inline void g()
{
  int a[f(1)];  // size dependent on the evaluation semantic of contract_assert above
}
```

— *end example*]

— *end note*]

9 When the program is *contract-terminated*, it is implementation-defined (depending on context) whether

— `std::terminate` is called,

— `std::abort` is called, or

— execution is terminated.

[*Note*: No further execution steps occur (6.10.2.3[intro.progress]). — *end note*]

[*Note*: Performing the actions of `std::terminate` or `std::abort` without actually making a library call is a conforming implementation of contract-termination (4.1.2[intro.abstract]). — *end note*]

10 If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is quick-enforce, the program is contract-terminated.

11 If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is enforce, observe, `noexcept-`enforce, or `noexcept-`observe, the contract-violation handler (6.11.3[basic.contract.handler]) is invoked with an lvalue referring to an object `v` of type `const std::contracts::contract_violation` (17.10.3[support.contract.violation]) containing information about the contract violation. Storage for `v` is allocated in an unspecified manner except as noted in 6.8.6.5.2[basic.stc.dynamic.allocation]. The lifetime of `v` persists for the duration of the invocation of the contract-violation handler.

12 If the contract violation occurred because the evaluation of the predicate exited via an exception, the contract-violation handler is invoked from within an active implicit handler for that exception (14.4[except.handle]). If the contract-violation handler returns normally and the evaluation semantic is observe, that implicit handler is no longer considered active.

[*Note*: The exception can be inspected or rethrown within the contract-violation handler. — *end note*]

13 If the contract-violation handler returns normally and the evaluation semantic is enforce or `noexcept`-enforce, the program is contract-terminated; if violation occurred as the result of an uncaught exception from the evaluation of the predicate, the implicit handler remains active when contract termination occurs.

Add a paragraph to and modify section 6.11.2[basic.contract.eval], paragraphs 16-17:

16 [*Note 4*: The terminating semantics terminate the program if execution would otherwise continue normally past a contract violation: the enforce ~~semantic provides~~and `noexcept`-enforce semantics provide the opportunity to log information about the contract violation before terminating the program or to throw an exception to avoid termination, and the quick-enforce semantic is intended to terminate the program as soon as possible as well as to minimize the impact of contract checks on the generated code size. Conversely, the observe ~~semantic~~and `noexcept`-observe semantics ~~provides~~provide the opportunity to log information about the contract violation without having to terminate the program. — *end note*]

16+a If a contract-violation handler invoked from the evaluation of a contract assertion exits via an exception, and the evaluation semantic is a nonthrowing semantic, the function `std::terminate` is invoked (14.6.2[except.terminate]).

17 If a contract-violation handler invoked from the evaluation of a function contract assertion (9.4.1[dcl.contract.func]) exits via an exception, and the evaluation semantic is not a nonthrowing semantic, the behavior is as if the function body exits via that same exception.

[*Note 5*: A *function-try-block* (14.1[except.pre]) is the function body when present and thus does not have an opportunity to catch the exception. If the function has a non-throwing exception specification, the function `std::terminate` is invoked (14.6.2[except.terminate]). — *end note*]

[*Note 6*: If a contract-violation handler invoked from an *assertion-statement* (8.9[stmt.contract.assert]) exits via an exception, and the evaluation semantic is not a nonthrowing semantic, the search for a handler continues from the execution of that statement. — *end note*]

14 **Exception handling** [except]

14.6 **Special functions** [except.special]

14.6.2 **The `std::terminate` function** [except.terminate]

Modify section 14.6.2[except.terminate], paragraph 1:

1 Some errors in a program cannot be recovered from, such as when an exception is not handled or a `std::thread` object is destroyed while its thread function is still executing. In such cases, the function `std::terminate` (17.9.5[exception.terminate]) is invoked.

[*Note 1*: These situations are:

(1.1) — when the exception handling mechanism, after completing the initialization of the exception

object but before activation of a handler for the exception (14.2[except.throw]), calls a function that exits via an exception, or

(1.2) — when the exception handling mechanism cannot find a handler for a thrown exception (14.4[except.handle]), or

(1.3) — when the search for a handler (14.4[except.handle]) exits the function body of a function with a non-throwing exception specification (14.5[except.spec]), including when a contract-violation handler invoked from an evaluation of a function contract assertion (6.11.2[basic.contract.eval]) associated with the function exits via an exception, or

(1.3+a) — when the search for a handler (14.4[except.handle]) exits a contract-violation handler invoked during the evaluation of a contract assertion with a nonthrowing evaluation semantic (6.11.2[basic.contract.eval]), or

(1.4) — when the destruction of an object during stack unwinding (14.3[except.ctor]) terminates by throwing an exception, or

(1.5) — when initialization of a non-block variable with static or thread storage duration (6.10.3.3[basic.start.dynamic]) exits via an exception, or

(1.6) — when destruction of an object with static or thread storage duration exits via an exception (6.10.3.4[basic.start.term]), or

(1.7) — when execution of a function registered with `std::atexit` or `std::at_quick_exit` exits via an exception (17.5[support.start.term]), or

(1.8) — when a *throw-expression* (7.6.18[expr.throw]) with no operand attempts to rethrow an exception and no exception is being handled (14.2[except.throw]), or

(1.9) — when the function `std::nested_exception::rethrow_nested` is called for an object that has captured no exception (17.9.8[except.nested]), or

(1.10) — when execution of the initial function of a thread exits via an exception (32.4.3.3[thread.thread.constr]), or

(1.11) — for a parallel algorithm whose `ExecutionPolicy` specifies such behavior (26.3.6.3[execpol.seq], 26.3.6.4[execpol.par], 26.3.6.5[execpol.parunseq]), when execution of an element access function (26.3.1[algorithms.parallel.defns]) of the parallel algorithm exits via an exception (26.3.4[algorithms.parallel.exceptions]), or

(1.12) — when the destructor or the move assignment operator is invoked on an object of type `std::thread` that refers to a joinable thread (32.4.3.4[thread.thread.destr], 32.4.3.5[thread.thread.assign]), or

(1.13) — when a call to a `wait()`, `wait_until()`, or `wait_for()` function on a condition variable (32.7.4[thread.condition.condvar], 32.7.5[thread.condition.condvarany]) fails to meet a postcondition, or

(1.14) — when a callback invocation exits via an exception when requesting stop on a `std::stop_source` or a `std::inplace_stop_source` (32.3.5.3[stopsource.mem], 32.3.9.3[stopsource.inplace.mem]), or in the constructor of `std::stop_callback` or `std::inplace_stop_callback` (32.3.6.2[stopcallback.cons], 32.3.10.2[stopcallback.inplace.cons]) when a callback invocation exits via an exception, or

(1.15) — when a `run_loop` object is destroyed that is still in the `running` state (33.12.1[exec.run.loop]), or

(1.16) — when `unhandled_stopped` is called on a `with_awaitable_senders<T>` object (33.13.2[exec.with.awaitable.senders]) whose continuation is not a handle to a coroutine whose promise type has an `unhandled_stopped` member function, or

(1.17) — when an object `scope` of type `std::execution::simple_counting_scope` or `std::execution::counting_scope` is destroyed and `scope.`*`state`* is not equal to *`joined`*, *`unused`*, or *`unused-and-closed`* (33.14.2.2.2[exec.simple.counting.ctor]), or

(1.18) — when `std::execution::get_parallel_scheduler` is called and `std::execution::parallel_scheduler_replacement::query_parallel_scheduler_backend()` returns a null pointer value (33.15[exec.par.scheduler]), or

(1.19) — when an exception is thrown from a coroutine `std::execution::task` (33.13.6[exec.task]) which doesn't support a `std::execution::set_error_t(std::exception_ptr)` completion.

— *end note*]

15 Preprocessing directives [cpp]

15.12 Predefined macro names [cpp.predefined]

Modify subclause 15.12[cpp.predefined], Table 22 [cpp.predefined.ft]:

Table 1: Table 22 — Feature-test macros [tab:cpp.predefined.ft]

Macro name	Value
⋮ *20 rows elided* ⋮	
`__cpp_constinit`	201907L
`__cpp_contracts`	202502L
`__cpp_contracts_nonthrowing_semantics`	XXXXXXL
`__cpp_decltype`	200707L
`__cpp_decltype_auto`	201304L
⋮ *56 rows elided* ⋮	

17 Language support library [support]

17.10 Contract-violation handling [support.contract]

17.10.1 Header `<contracts>` synopsis [contracts.syn]

Modify section 17.10.1[contracts.syn], paragraph 1:

1 The header `<contracts>` defines types for reporting information about contract violations (6.11.2[basic.contract.eval]).

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

  enum class assertion_kind : unspecified {
```

```
    pre = 1,
    post = 2,
    assert = 3
  };

  enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4,
    noexcept_observe = 5,
    noexcept_enforce = 6
  };

  enum class detection_mode : unspecified {
    predicate_false = 1,
    evaluation_exception = 2
  };

  class contract_violation {
    // no user-accessible constructor
  public:
    contract_violation(const contract_violation&) = delete;
    contract_violation& operator=(const contract_violation&) = delete;

    see below ~contract_violation();

    const char* comment() const noexcept;
    contracts::detection_mode detection_mode() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

17.10.2 **Enumerations** **[support.contract.enum]**

Modify section 17.10.2[support.contract.enum], Table 45 [support.contract.enum.semantic]:

Table 45 — Enum `evaluation_semantic` **[tab:support.contract.enum.semantic]**

Name	Meaning
`ignore`	Ignore evaluation semantic
`observe`	Observe evaluation semantic
`enforce`	Enforce evaluation semantic
`quick_enforce`	Quick-enforce evaluation semantic
`noexcept_observe`	`noexcept`-observe evaluation semantic
`noexcept_enforce`	`noexcept`-enforce evaluation semantic

6 Conclusion

This small addition to C++26 Contracts will enable fine-grained control over the support for exceptions, allowing people to mitigate many of the concerns that have been expressed with throwing

violation handlers.

Acknowledgments

Thanks to Ville Voutilainen for persisting in pursuit of nonthrowing evaluation semantics.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of the prototype implementations.

Bibliography

[P3100R7] Timur Doumler and Joshua Berne, "A framework for systematically addressing undefined behaviour in the C++ Standard", 2026
http://wg21.link/P3100R7

[P3290R6] Joshua Berne, Timur Doumler, and John Lakos, "Integrating Existing Assertions With Contracts", 2026
http://wg21.link/P3290R6

[P3400R4] Joshua Berne, "Controlling Contract-Assertion Properties", 2026
http://wg21.link/P3400R4

[P3850R1] Timur Doumler and Joshua Berne, "A proposed plan for extending Contracts in C++29", 2026
http://wg21.link/P3850R1