Remove `evaluation_exception()` from contract-violation handling for C++26

Peter Bindels (dascandy@gmail.com)
Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)
Eric Fiselier (eric@efcs.ca)
Iain Sandoe (iain@sandoe.co.uk)

Document #:	P3819R0
Date:	2025-09-02
Project:	Programming Language C++
Audience:	LEWG

Abstract

For C++26, we propose to remove the member function `evaluation_exception()` from the type `std::contracts::contract_violation`. On some platforms, an implementation of this function may require user code execution after a contract violation but before the contract-violation handler, which may pose a security risk. We can add this function back in future versions of C++ if it can be shown that the security risk is fully avoidable. In the meantime, the functionality it offers is available through other means.

1 The status quo

The standard library API for contract-violation handling, added to the C++26 working draft [N5014] via [P2900R14], contains the type `std::contracts::contract_violation`. When the program has a user-defined contract-violation handler, and a contract violation occurs, the implementation constructs an object of this type and passes a reference to it into the contract-violation handler.

The purpose of the type `std::contracts::contract_violation` is to provide access to information about the contract violation that occurred. This includes the source location of the violated assertion and the failure mode: did the contract predicate evaluate to `false`, or did evaluation of the predicate exit via an exception? All information is provided via a set of `const` member functions.

One of these member functions is `evaluation_exception()`. It is specified as follows:

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

If the contract violation occurred because evaluation of the predicate exited via an exception, this function returns an `exception_ptr` to that exception; otherwise, it returns a null pointer.

The functionality offered by `evaluation_exception()` is also available through other means. To determine whether evaluation of the predicate exited via an exception, we can check whether a call to the member function `detection_mode()` returns the enum value `evaluation_exception`. If it does, we can call `std::current_exception()` to obtain an `exception_ptr` to that exception:

```
void handle_contract_violation (const contract_violation& cv) {
  if (cv.detection_mode() == detection_mode::evaluation_exception) {
    auto evaluation_exception_ptr = std::current_exception();
    // handle
}
```

Further, to handle exceptions of a particular type, we can re-throw the current exception in the contract-violation handler and immediately catch it (this technique is called a Lippincott function):

```
void handle_contract_violation (const contract_violation& cv) {
  if (cv.detection_mode() == detection_mode::evaluation_exception) {
    try {
      throw;
    } catch (std::exception& e) {
      // handle
    }
}
```

However, these techniques are more verbose and require more care to use. If no exception was thrown during the contract check, but the user forgets to query `detection_mode()`, they will either fail (if no exception is currently being handled), or worse get an exception that is currently being handled but that was not the cause of the contract violation (this is possible if the contract violation occurred within a `catch` clause handling some other exception). The member function `evaluation_exception()` was intended to give the user a simple and direct way to access the exception thrown during the contract check without having to worry about any other exceptions.

2 The problem

When `evaluation_exception()` was proposed in [P3227R1] and added to [P2900R14], there was an assumption that would essentially be syntactic sugar for the simple logic above: check `detection_mode()` and invoke `std::current_exception()` if that mode is `evaluation_exception`. However, a problem arises when an exception is thrown within a contract-violation handler and `evaluation_exception()` is then invoked within the `catch` clause handling that internal exception:

```
void handle_contract_violation (const contract_violation& cv) {
  // ...
  try {
    // ...
    throw X;
  } catch (...) {
      if (cv.detection_mode() == detection_mode::evaluation_exception) {
        // the current exception is now X, not whatever was thrown by the contract check!

        auto evaluation_exception_ptr = cv.evaluation_exception();
        // handle the original exception, not X
      }
  }
}
```

In order to always return the exception that originated from evaluating the violated contract assertion, even inside such a `catch` clause within the contract-violation handler, the implementation must do extra work to preserve access to that exception before invoking the contract-violation handler. In particular, since the adoption of [P2900R14] for C++26 we have learned that on some platforms, this requires *copying* the exception object *before* invoking the contract-violation handler.

Requiring implementations to perform such a copy is problematic as it can lead to *user code* being executed after a contract violation has been detected but before the associated contract-violation handler is called. This, in turn, may pose a security risk (see also [P3417R1]).

When a contract violation has been detected, the program may be in an invalid state, for example a corrupted stack. The contract-violation handler is user code, but it is user code that is expected to be run in such circumstances, and can be written to be robust against them. On the other hand, the copy constructor of an arbitrary exception type will typically not be written with such robustness in mind. For example, it could walk the stack (one might want to save the stack trace at the time when the exception object was created or copied). This opens up a security vulnerability: an attacker could corrupt the stack and then use the exception copy constructor to jump to an arbitrary place and execute arbitrary code.

On platforms implementing the Itanium ABI (GCC, Clang), there is a known implementation strategy for `evaluation_exception()` that avoids such a copy of the exception object. However, for the Microsoft ABI, we were unfortunately so far unable to confirm the existence of such an implementation strategy. In the general case, the C++ exception API does not yet expose a way to implement this feature without potentially making a copy. Thus, implementing this function would put a new requirement on all future exception-handling implementations.

3 Proposal

The contract-violation handling mechanism in the C++26 working draft (as originally proposed in [P2811R7], which did not yet contain `evaluation_exception()`) was carefully designed to avoid potential security risks. To this end, it consciously avoided executing user-defined code or mandating any operations that might be overly non-trivial after a contract violation has occurred but before the call to the contract-violation handler.

Unless and until we can be sure that `evaluation_exception()` is implementable on all platforms without violating this design principle, we should not ship it in a C++ Standard. We therefore propose to remove the member function `evaluation_exception()` entirely before shipping C++26. This function can easily be added back in future versions of C++ when we have a better understanding of the potential security risks.

In the meantime, users can access an exception thrown from a contract check through other means, such as by using `std::current_exception()` and Lippincott functions. As discussed above, these techniques have known gotchas, but they can be used correctly and efficiently and do not come with security concerns. On some platforms, accessing the exception may still incur a copy, but that copy happens inside the user-defined contract-violation handler, where the user has full control over it.

4 Wording

The proposed wording is relative to [N5014].

Modify the header `<contracts>` synopsis ([contracts.syn]) as follows:

```
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;
  detection_mode detection_mode() const noexcept;
  ~~exception_ptr evaluation_exception() const noexcept;~~
  bool is_terminating() const noexcept;
```

```
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
};
```

Remove the following paragraph from [support.contract.violation]:

~~`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` object that refers to that exception or a copy of that exception; otherwise, a null `exception_ptr` object.~~

References

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

[P2811R7] Joshua Berne. Contract-violation handlers. https://wg21.link/p2811r7, 2023-06-27.

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r14, 2025-02-13.

[P3227R1] Gašper Ažman and Timur Doumler. Fixing the library API for contract violation handling. https://wg21.link/p3227r1, 2025-02-27.

[P3417R1] Gašper Ažman and Timur Doumler. Handling exceptions thrown from contract predicates. https://wg21.link/p3417r1, 2025-02-27.