Handling exceptions thrown from contract predicates

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

Document #: P3417R1
Date: 2025-02-27
Project: Programming Language C++
Audience: SG21, EWG

Abstract

In this paper, we explore a modification to how exceptions thrown from the evaluation of a contract predicate are handled. In particular, we consider handling such an exception separately from an exception that was being handled at the time when the contract violation occurred. In this model, `throw;` would only rethrow the latter, never the former; `std::current_exception()` would return a pointer only to the latter, never to the former; and `std::contracts::contract_violation::evaluation_exception()` would return a pointer to the former, never to the latter. Such a change would simplify the API and remove a potential footgun. However, after considering the three known specification and implementation strategies for this change, we conclude that their tradeoffs are ultimately unfavourable, and therefore do not propose any change to the status quo.

Revision history

R0 (October 2024 mailing):

- Original version of the paper

R1 (March 2025 mailing):

- No longer proposing a change. The paper is now purely informational; this revision is being published to preserve the history of the discussion
- Incorporated feedback from R0
- Various editorial edits

1 The problem

Contracts for C++, as adopted into the C++26 working paper via [P2900R14], specify that when the evaluation of a contract predicate exits via an exception, the contract-violation handler is called and acts as an exception handler for that exception. Therefore, the exception can be retrieved with the following incantation:

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

where `my::handle` is a user-defined handler that takes a `std::exception_ptr`.[1] This incantation is rather unwieldy, so we adopted [P3227R1], which adds a member function `evaluation_exception()` to the class `contract_violation`, simplifying the above code to the much more user-friendly:

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

This member function `evaluation_exception()` behaves as follows:

- If the contract violation occurred because the contract predicate evaluation exited via an exception, `evaluation_exception()` will return a pointer to *that* exception;
- Otherwise, `evaluation_exception()` will return null.

However, this addition to the Contracts library API does not alter the pre-existing behaviour of `std::current_exception()`. As a result, `std::current_exception()` may return a pointer to two very different flavours of exception, depending on the program state:

- If the contract violation occurred because the contract predicate evaluation exited via an exception, `std::current_exception()` will return a pointer to *that* exception, regardless of whether the contract violation occurred while some other exception was being handled;
- If the contract violation occurred because the contract predicate evaluated to `false` while some exception was being handled, i.e., inside a `catch` clause (which could be multiple stack frames further up), `std::current_exception()` will instead return a pointer to *that* exception;
- Otherwise, `std::current_exception` will return null.

In other words, `evaluation_exception()` and `std::current_exception()` may or may not point to the same exception. This somewhat surprising behaviour is the consequence of treating the predicate evaluation exception as just another exception on the exception stack, in the same way as if it had originated from any other part of the program.

---

[1] In order to actually get to the exception object itself, such an `std::exception_ptr` would have to be rethrown and then caught with a catch-clause appropriate for the exception's type.

Further, consider what happens when we rethrow the current exception from the contract-violation handler:

```
void handle_contract_violation (contract_violation& violation) {
  if (auto ex = std::current_exception())
    std::rethrow_exception(ex);  // or just throw; – what happens now?
}
```

With the current specification, this may either rethrow the predicate evaluation exception or, if the contract check did not throw an exception but occurred inside a `catch` clause in user code, rethrow the entirely unrelated exception that was being handled there.

In many cases, the user might want to handle those different flavours of exceptions differently. For example, when installing a throwing contract-violation handler, the user might want to rethrow the exception that was thrown by the predicate and have code further up the stack that can handle it; the archetypical example is a predicate evaluation that throws `std::bad_alloc`. At the same time, it might not make sense to rethrow an exception from the contract-violation handler that is entirely unrelated to the contract check. It is possible to distinguish the two flavours via `evaluation_exception()`, but the fact that the user can also get to the exception via the more familiar `std::current_exception()` API, which does not distinguish the two flavours at all, is a potential footgun.

2 Possible solution

In order to remove the footgun described above, we would have to change the semantics as follows. Both `throw;` and `std::current_exception()` should never refer to the predicate evaluation exception, but always to the exception that was being handled at the time when the contract violation occurred (if any). Independently from those facilities, the predicate evaluation exception would still be accessible in the contract-violation handler via the `std::contracts::contract_violation` member function `evaluation_exception()`, but that would now be the *only* way to access that exception:

C++26 Working Paper	**This paper**

`bool pred() { throw 666; }`
`void f() pre (pred());`
 	
`int main() {`
`  try { throw 777; }`
`  catch (...) { f(); }`
`}`
 	
`void handle_contract_violation`
`(const contract_violation& v) {`
`  auto p1 = v.evaluation_exception();`
`  // p1 points to 666`
 	
`  auto p2 = std::current_exception();`
`  // p2 points to 666`
`}`
	`bool pred() { throw 666; }`
	`void f() pre (pred());`
	
	`int main() {`
	`  try { throw 777; }`
	`  catch (...) { f(); }`
	`}`
	
	`void handle_contract_violation`
	`(const contract_violation& v) {`
	`  auto p1 = v.evaluation_exception();`
	`  // p1 points to 666`
	
	`  auto p2 = std::current_exception();`
	`  // p2 points to 777`
	`}`

Such semantics have the property that the C++ language would treat exceptions thrown during contract checks as entirely separate from the remainder of the program, and any other exceptions being handled in that program, consistent with the design principles in [P2900R14] stipulating that contract assertions are "ghost code".

3 Specification and implementation strategies

3.1 Separate exception stacks

The first and most radical strategy is to literally use a separate exception stack for exceptions thrown during contract checks, and to specify `evaluation_exception()` to refer to the top exception on that separate stack. While this would provide the cleanest and conceptually simplest separation between the different flavours of exception, it would also be a substantial change to the exception-handling machinery in current compilers – and possibly unfeasible on at least some of them – as well as an ABI break.

3.2 Handle exception before the contract-violation handler

Unlike the first strategy, the second strategy does not require any changes to the underlying exception-handling machinery in C++. We can express this strategy as a modification of the pseudocode in [P2900R14], Section 3.5.10 that illustrates the compiler-generated contract-violation handling process (simplified to show only the parts relevant for the *observe* and *enforce* evaluation semantics that may result in a call to the contract-violation handler):

P2900R14	**This paper**

`bool _violation;`
`bool _handled = _mode;`
`detection_mode _dm;`
 	
`try {`
`  _violation = !predicate;`
`}`
`catch (...) {`
`  _violation = true;`
`  _mode = evaluation_exception;`
`  handle_contract_violation(...);`
`  _handled = true;`
`}`
 	
`if (_violation && !_handled) {`
`  _mode = predicate_false;`
`  handle_contract_violation(...);`
`}`
	`bool _violation;`
	`detection_mode _mode;`
	`std::exception_ptr _evaluation_exptr;`
	
	`try {`
	`  _violation = !predicate;`
	`  _mode = predicate_false;`
	`}`
	`catch (...) {`
	`  _violation = true;`
	`  _mode = evaluation_exception;`
	`  _evaluation_exptr =`
	`  std::current_exception();`
	`}`
	
	`if (violation) {`
	`  handle_contract_violation(...);`
	`}`

Instead of specifying that the contract-violation handler is called within an implicit handler for the exception thrown during predicate evaluation, we consider such an exception to be handled *before* the contract-violation handler is called. As a result, the exception is no longer on the exception stack when the contract-violation handler is called, and can be accessed only via `evaluation_exception()` but not via `throw;` or `std::current_exception()`.

However, the tradeoff is that the implementation now needs to somehow save an exception thrown during predicate evaluation past its handling lifetime, as the exception needs to survive the closing brace of its `catch` handler without being the active exception so it can be made accessible in the contract-violation handler via `evaluation_exception()`. This would typically happen as if by creating a `std::exception_ptr` to that exception, as reflected in the pseudocode above. If this copy itself throws an exception, `evaluation_exception()` will instead contain *that* exception, or `std::bad_exception`, consistent with how `std::make_exception_ptr` works in C++ today.

On platforms implementing the Itanium ABI (GCC, Clang), this strategy is straightforward as exceptions are allocated on the heap; creating a `std::exception_ptr` pointing to an active exception as above involves little more than incrementing a refcount, and the required lifetime extension happens automatically. However, on MSVC, exceptions are kept on the stack; creating a `std::exception_ptr` to an active exception requires allocating dynamic memory and copying the exception object into that memory. The consequence is that the copy constructor of the thrown exception – i.e., *user-defined code* – will be called *after* the contract violation has occurred but *before* its associated contract-violation handler is called. This runs afoul of an important design principle in the current specification of Contracts for C++ to never run user-defined code within that gap.

The current specification carefully avoids doing so: after a contract violation was detected, the implementation will create a `contract_violation` object, which just loads some static data into a small struct on the stack, and then immediately calls the contract-violation handler. That handler is user-defined code, but it is user-defined code that is expected to be run when the program is in an invalid state, for example a corrupted stack, and can be written to be robust against such circumstances.

On the other hand, the copy constructor of an arbitrary exception type will typically not be written with such robustness in mind; it could walk the stack (for example, one might want to save the stack trace at the time when the exception object was created or copied), which in the face of a corrupted stack might create a security risk (an attacker could corrupt the stack and then use the exception copy constructor to jump to an arbitrary place and execute arbitrary code). This security risk is the reason why, when the current contract-violation handling mechanism was adopted into Contracts for C++ via [P2811R7], executing user-defined code or mandating any operations that might be overly non-trivial before the call to the contract-violation handler was consciously avoided.

3.3 Tweak the exception handling mechanism

Unfortunately, any solution that portably avoids copying the exception before calling the contract-violation handler invariably requires *some* changes to the existing C++ exception handling mechanism. To keep these changes minimal compared to the first strategy, we could contemplate a third strategy: keep the exception thrown from a contract predicate evaluation on the normal exception stack, but modify the user-facing facilities providing access to that exception stack, in particular `throw;` and `std::current_exception()`, such that they refer to the second exception on the stack in case the top exception on the stack is the one thrown from a contract predicate evaluation.

However, it is currently unclear whether such a strategy is logically sound and/or actually avoids copying the predicate exception in all cases. The exception stack can contain multiple exceptions thrown from a contract predicate evaluation, and moreover, such exceptions can be *interspersed* with exceptions unrelated to contract checking.

For example, consider the case where, either during predicate evaluation or during execution of the contract-violation handler, some function (unaware that it is called in that context) throws an exception unrelated to contract checking, and that exception is handled in an intermediate `catch` clause before continuing. Within such a `catch` clause, the usual exception-handling facilities such as `throw;` and `std::current_exception()` all need to have the same semantics as usual in order for such handling to succeed. It is currently unclear how to achieve the required semantics with the third strategy.

4 Summary

In the current specification of Contracts for C++, `evaluation_exception()` provides a way to access an exception thrown during evaluation of a contract predicate from within the associated contract-violation handler. At the same time, pre-existing C++ exception-handling facilities such as `throw;` and `std::current_exception()` may also refer to that same exception, or may instead refer to an unrelated exception that was being handled when the contract violation occurred.

From a language design perspective, it seems desirable to remove this ambiguity of the latter facilities and to treat exceptions thrown during contract checking entirely separate from exceptions thrown elsewhere. However, after considering all three known specification strategies for proposing such a change to the C++26 working paper, we found that none of them are viable: they are either incompatible with current C++ toolchains, add security risks that may prove unacceptable, or do not seem logically consistent. We therefore conclude that leaving the current specification of Contracts for C++ as-is is the lesser evil.

Acknowledgements

Many thanks to Eric Fiselier, Ville Voutilainen, Joshua Berne, and Iain Sandoe for their valuable feedback on the material discussed in this paper.

References

[P2811R7] Joshua Berne: "Contract-Violation Handlers". 2023-06-27

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-02-13

[P3227R1] Gašper Ažman and Timur Doumler: "Fixing the library API for contract violation handling". 2024-10-24