Throwing from a `noexcept` function should be a contract violation.

Document #:	D3205R0
Date:	2024-03-27
Project:	Programming Language C++
Audience:	SG21, EWG, LEWG

Reply-to:	Gašper Ažman <gasper.azman@gmail.com> Jeff Snyder <jeff@caffeinated.me.uk> Andrei Zissu <andrziss@gmail.com>

Contents

1 Introduction

Throwing an exception from a noexcept function currently has the effect of calling `std::terminate()`. This has given rise to the Lakos rule, over which many, many papers have been written.

This paper proposes to resolve the debate.

We contend that the call to `std::terminate()` was merely the best we could do at the time to enforce the meaning of `noexcept`, and we can do better now that we have contract violation handlers.

We propose to change the effect of throwing from a `noexcept` function be a function postcondition violation, have configurable semantics as per the Contracts MVP ([P2900R5]), and therefore, in `observe` and `enforce` modes, call the violation handler.

2 Proposal

We propose that throwing from a `noexcept` function be treated as a violation of a *postcondition assertion*, instead of unconditionally calling `std::terminate()`.

This postcondition would be the "last" (outermost) postcondition to evaluate, so any exceptions that a violation handler throws due to other postconditions still hit the `noexcept` barrier.

3 Rationale

The Lakos rule is there because testing defensive precondition checks in function implementations by throwing exceptions is useful to a part of the C++ developer community. `noexcept` functions effectively make this technique impossible.

`noexcept` is reflectable because it matters for exception safety. It means that the function will not allow an exception to escape when called in-contract, and thus there is no need to choose a more expensive algorithm to achieve exception guarantees in the presence of possible exception throws.

This property is important in

— move, construct, and destroy operations.
— async callbacks, where stack unwinding out of the callback would proceed to the runloop instead of being propagated to the continuation.
— correctness of exception-unsafe code that wants to ensure some dependency-injected component won't jeoperdise its correctness.

The `std::terminate()` semantic is unlikely to be relied upon as a matter of deliberate control flow. It is quite clearly a stand-in for a postcondition violation; people do rely on exit handlers for recovery `std::terminate` happens to get called because of a bug - but it seems doubtful that someone would rely on an exception calling `std::terminate()` instead of calling `std::terminate()` explicitly.

If we instead redefine throwing from a `noexcept` function as a contract violation, a violation handler could instead just let the exception propagate and unwind, achieving the goal of negative testing, while still allowing the required reflectable properties for code not under test.

3.1 Example: negative testing through `noexcept` boundaries

In a unit test, one might install the following handler:

```
void handle_contract_violation(contract_violation const& violation) {
    if (violation.detection_mode() != evaluation_exception) {
        // 1 - the precondition to be checked will emit this
        throw MyTestException(violation);
    }
    // 2 - if the exception-in-flight is the one we just threw in (1)
    try {
        throw;
    } catch (MyTestException const&) { // it's a test exception
        throw; // rethrow it
    } catch (...) {
        // for other exceptions, noexcept is still noexcept
        invoke_default_violation_handler(violation);
    }
}
```

The Lakos rule then has no further reason to exist, and we can use `noexcept` to mean a reflectable postcondition of "function does not throw when called in-contract" freely.

4 `noexcept` and `[[throws_nothing]]`

`noexcept` is a reflectable property that *also* places a postcondition of not throwing on the function.

`[[throws_nothing]]` is the [P2946R1]-proposed syntax of also placing such a post-condition on a function, but with a default of the `ignore` semantic.

We should really unify this universe of exceptionless postconditions.

5 Viability of negative testing

Negative testing has to be done very carefully - after all, the test program deliberately calls the function-under-test out-of-contract.

As an example, code that is exception-unsafe cannot be negative-tested using exceptions.

```
void wrapper1(std::function<void>()noexcept f)
{
    std::lock_guard g(some_lock);
    ...
    std::unlock_guard(g);
     wrapper2(f);
}

void wrapper2(std::function<void>()noexcept f)
{
    some_lock.lock();
    f();
    some_lock.unlock();
}

wrapper1([]pre(false){}); // deadlocks
```

```
std::mutex r;

template <typename F>
void with_something(F f) noexcept requires(noexcept(f()))
{
    r.lock();
    f();
    r.unlock()
}
```

Negative-testing `f()` *through* `with_something` will deadlock the next test. Note, however, that `f()` is deliberately invoked out-of-contract, and therefore already requires extreme care. Having some tests is better than having none, so this proposal still leaves the engineer in a more capable position.

6 ABI considerations

6.1 Is this an ABI break?

Not… necessarily. Yes. But not a bad one.

Of course, we are changing semantics. However, `std::terminate` is a valid implementation of an implementation-defined handler for a specific violation semantic, so technically today's code is conforming.

The change comes if you set your own violation handler and set the semantic (in an implementation-defined way) to not be the kind that calls `std::terminate()`, but at that point you're already recompiling your code.

6.2 Can I link with past code?

Su

7 Prior art

— [N3248] discusses the reasons we need the Lakos rule, which are obviated by the proposed change
— [P1656R2] discusses the actual desires of annotating functions that are prevented by the Lakos rule
— [P2837R0] discusses why we need the Lakos rule
— [P2900R5] is the current contracts proposal
— [P2946R1] says that `[[throws_nothing]]` could imply a contract violation on throwing
— [P3155R0] proposes the application of the Lakos rule in the standard library
— [P3085R0] has a similar conception of what `noexcept` means.

8 FAQ

— If an assertion fails due to the predicate exiting via exception, the violation handler throws, does that exception still hit the function's `noexcept` barrier and `std::terminate()`?

Yes.

9 Acknowledgements

— Jonas Persson contributed comments with regards to noexcept destructors, double throws, and unwinding code generation overhead.

10 References

[N3248] A. Meredith, J. Lakos. 2011-02-28. noexcept Prevents Library Validation.
https://wg21.link/n3248

[P1656R2] Agustín Bergé. 2020-02-14. "Throws: Nothing" should be noexcept.
https://wg21.link/p1656r2

[P2837R0] Alisdair Meredith, Harry Bott. 2023-05-19. Planning to Revisit the Lakos Rule.
https://wg21.link/p2837r0

[P2900R5] Joshua Berne, Timur Doumler, Andrzej Krzemieński. 2024-02-15. Contracts for C++.
https://wg21.link/p2900r5

[P2946R1] Pablo Halpern. 2024-01-16. A flexible solution to the problems of 'noexcept'.
https://wg21.link/p2946r1

[P3085R0] Ben Craig. 2024-02-10. 'noexcept' policy for SD-9 (throws nothing).
https://wg21.link/p3085r0

[P3155R0] Timur Doumler, John Lakos. 2024-02-15. noexcept policy for SD-9 (The Lakos Rule).
https://wg21.link/p3155r0