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

Document #:	D3205R0
Date:	2024-03-31
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

At present (C++23), throwing an exception from a `noexcept` function has the effect of calling `std::terminate()`.

Throwing from a `noexcept` function is clear erroneous behaviour. It seems to follow that the current behavior is a stand-in for "don't do that", as well as a last-resort effort to prevent actually breaching the `noexcept` part of the function contract.

There is a concerted effort underway in WG21 to both classify and figure out what to do with erroneous behavior in general:

— Thomas Koppe's [TODO PAPER REF] introduces the concept of Erroneous behaviour into the language.

— The entire Contracts facility ([P2900R5] and associated papers from SG21) could be seen as a way to let users define and catch erroneous behaviour at library interfaces.

This paper brings breaching the `noexcept` specification of a function into the same discussion; and, since the Contracts facility aims to let users configure what to do when Erroneous behavior happens, allow novel ways of handling this eventuality.

We propose to change the effect of throwing from a `noexcept` function be Erroneous Behaviour, and as such a function "epilogue" check failure (it's not a post-condition failure, as post-conditions are only guaranteed on nonexceptional function exit).

We also propose it to have a configurable semantic as per the Contracts MVP ([P2900R5]). We propose a new semantic that is equivalent to the C++23 status-quo.

2 Definitions

As per Lisa Lippincott's work on function interfaces [TODO CITATION], a function's contract checking interface comprises a function *prologue* and a function *epilogue*.

Example, in pseudocode, with 2 extra keywords:

— `result`, an alias for the return object
— `implementation;` a statement denoting the function body, including the `try`-block, if present.

```
int halve(int x)
  interface {
    // begin prologue
    try {                       // T1
      contract_assert(x > 0); // PRE
      auto old_x = x;           // POST1
    // end prologue
      implementation; /* does not see old_x */
    // begin epilogue
      contract_assert(old_x < result); // POST2
    } catch (...) {         // T2
        std::terminate(); // T3
    }                       // T4
    // end epilogue
  }
/* implementation */ {
    return x / 2;
}
```

The function **prologue** (PRE, POST1, T1) notionally runs after the function parameters have been bound to function arguments and before entering the function body (implementation); it is here that, for instance, `pre`-condition assertions are checked, and data needed to check function postconditions is captured.

As an example, line (PRE) is equivalent to the [P2900R5] `pre (x > 0)`, if we apply the `pre` assertion flavor to it.

The function **epilogue** (POST1, T2, T3, T4) notionally runs after the return value has been constructed and the function body has exited; all local variables have been destroyed, but the function arguments are still within lifetime. It is here that postconditions are checked, for instance.

As an example, lines (POST, POST2) are equivalent to [TODO CAPTURES CITATION] `post [old_x = x] (r: old_x < r)`, if we apply the `post` assertion flavor to (POST2).

The construct in lines (T1, T2, T3, T4) is equivalent to the [**2946R1?**] `[[throws_nothing]]` (with the `ignore` semantic), and would be equivalent to `noexcept` if it were reflectable (with the `noexcept()` query).

We should really unify this universe of exceptionless postconditions.

3 Proposed semantics

3.1 Semantics

We propose that throwing from a `noexcept` function be treated as a violation of a contract assertion in the function epilogue, instead of unconditionally calling `std::terminate()`. In terms of the example in the `definitions`, to change

Current

```
int halve(int x)
  interface {
    try {
      implementation;
    } catch (...) {
      std::terminate();

    }
  }
{ return x/2; }
```

Proposed

```
int halve(int x)
  interface {
    try {
      implementation;
    } catch (...) {
      contract_assert(false);
      throw;
    }
  }
{ return x/2; }
```

3.2 A new assertion kind: `excspec`

`noexcept` and `[[throws_nothing]]`, as well as the deprecated exception specifications on functions, specify what kind, and whether, exceptions can be propagated from functions. Lewis Baker's paper [TODO FIND LEWIS PAPER CITATION] on allowable exception specifications is also proposing further additions in this space.

This paper proposes a new assertion kind, to be added to `pre`, `post`, and `assert`: `excspec`. It is the assertion kind that is used for exception specification failures; that is, whenever a function throws an exception that it is not declared to throw (`noexcept`, `[[throws_nothing]]` and `throw()` are all covered by this rule).

The reason to propose `excspec` and not merely `exc`, which would be used for all kinds of exception failures, including ones from future extensions that would allow checking exception *states* on exceptional exits, is that `catch` catches by *type*, which means that type failures are of a whole different nature (unwinding-wise) than program state failures.

We have not done detailed research on whether this is the correct choice; this is the initial recommendation of this paper, however.

3.3 A new check semantic: `terminate`

We propose a new semantic be added to the [P2900R5]-proposed semantics, with the provisional name `terminate`. A contract check failure with the `terminate` semantic is equivalent to a call to `std::terminate`.

If an implementation chooses to default to the `terminate` semantic for exception specifiation check failures such as this one, a C++23 implementation is already conforming with this proposal.

In the "matrix" of semantic properties, this is where the `terminate` semantic fits in: [TODO ANDREI ZISSU CITATION AND HARMONIZE MATRIX REPRESENTATION]

semantic	checks	calls handler	assumed after	terminates	proposed
assume	no	no	yes	no	no
ignore	no	no	no	no	[P2900R5]

semantic	checks	calls handler	assumed after	terminates	proposed
"Louis"	yes	no	yes	`trap`-ish	TODO
terminate	yes	no	yes	`std::terminate`	here
observe	yes	yes	no	no	[P2900R5]
ensure	yes	yes	yes	`std::abort`-ish	[P2900R5]

Note that in this table, *assumed after* depends on the semantic being fixed at compile-time. In general, the semantic is chosen per-evaluation, so it can be chosen at link-time or even run-time.

The "Louis" semantic, in particular, has barely any reason to exist if it is not fixed at compile-time (its main use-case is reducing code-bloat).

3.4 Recommended practice

The standard cannot mandate cross-translation-unit behavior or behavior of erroneous programs.

Nevertheless, there are clear design intentions which should be communicated. This section is for that.

3.4.1 Separate configurability

If configurable at all, implementations should offer configuration for exception specification check failures separately from other checks.

This is for two main reasons:

— `noexcept` is reflectable because it allows choosing a different algorithm while maintaining exception safety; stack unwinding when throwing from a `noexcept` function is very likely to run into further undefined or erroneous behavior.
— changing away from the status-quo in this case by "accident" when configuring other erroneous behavior checks seems user-hostile.

3.4.2 Default semantic

Implementations should use the `terminate` semantic by default on exception specification failures, and thus remain conforming by merely documenting this fact.

This avoids an ABI break but still provides for more use-cases than C++23 allows.

3.4.3 Source-location for `excspec` violations in `contract_violation` objects

— *If* the implementation chooses to make the semantic of `exspec` checks configurable
— *and if* the semantic chosen invokes the violation handler (`observe`, `ensure`)

then the implementation *may* choose to provide useful failure information through the `contract_violation` argument of the violation handler call. This may involve some program size expansion to store all the static data.

Note that [P2900R5] already makes the presence of such data implementation-defined. Also note that an implementation may choose to use emitted debug information, coupled with the stack trace, to obtain said data when and if the violation handler asks for it.

4 Discussion

An analysis of the behavior under different conditions, and what it may be good for.

4.1 Program flow under various semantics

… in which we discuss the behaviour of the proposal under various proposed semantics.

4.1.1 Default: behavior under the `terminate` semantic

If the semantic for `excspec` check failures is `terminate`, the behaviour is unchanged from C++23: `std::terminate` is called, and recovery is not possible without restarting the process.

The implementation may optimize based on the assumption of the function not throwing.

4.1.2 The "Louis" semantic

Under the "Louis" semantic (just use a trap instruction on check failure), currently used by *libc++* to implement cheap hardening checks, `noexcept` becomes even cheaper; the implementation is allowed to emit even shorter code, with no termination handler invocation.

The implementation may optimize based on the assumption of the function not throwing. This may be a good fit for low-resource applications.

4.1.3 The `enforce` semantic

Under the `enforce` semantic, the violation handler would get invoked, allowing the program to phone home, gather more data, and in general recover the same way as it does on all contract failures, which would be a welcome unification of handling of erroneous behavior, as opposed to the termination handler, which has a difficult time distinguishing between erroneous terminations and clean terminations.

The implementation still gets to optimize based on the assumption that the function cannot throw.

Compared to the "Louis" and `terminate` semantics, more static program data may be stored because of source location information.

4.1.4 The `observe` semantic

The `observe` semantic is *a lot* more interesting than the previous three - the program owner may now use the violation handler as a general-purpose recovery mechanism.

Under the `observe` semantic, the **implementation may not optimize** on the non-throwing nature of the function, but a library *may* via the `noexcept()` query. This is obviously an ABI break, so an implementation *may* choose to not offer such options, or refuse to link compilation units where this aspect differs (similarly to `-fPIC`).

If a `noexcept` function throws, the `observe` semantic on the check will invoke the violation handler, and then *continue*. The violation handler may now observe the failure, and decide what to do. There are use-cases for every possible decision, which are explored below.

4.1.4.1 Terminate the program

A violation handler could just call a termination function explicitly. This has the same effect as the terminating semantics, but without the allowance to optimize on nonthrowing behavior.

4.1.4.2 Rethrow the exception

This might be useful for negative testing, or last-ditch recovery when we don't need strong exception guarantees, but *do* need nontermination. This has been the argument in [TODO CITE BJARNE'S PAPER] - stack unwinding is sometimes still the best of bad options when we have erroneous behaviour, instead of termination.

This would also happen if the violation handler simply does nothing.

NOTE: this is consistent with allowing the program to proceed to the body of the function after a failed but ignored precondition check.

The authors are aware that ignoring check failures is generally a bad idea. Nevertheless, the `observe` *semantic is useful in transitional scenarios, so we chose to keep it; this is no different. Argue with the* `observe` *semantic.*

The primary use-case for this is newly marking functions `noexcept` or `[[throws_nothing]]` - one may log the exception and continue the program as-before.

4.1.4.3 Modify the exception

A violation handler could choose to throw a *different* exception. This may be used by the program-wide failure handler to do something useful, assuming nothing else catches the exception. It may also be used for negative testing, to rethrow *expected* exceptions back to the test driver.

That said, negative testing is probably better served with a separate facility (also see the section on negative testing).

4.1.5 The `ignore` semantic

Oh, the poster-child for seemingly horrible choices.

There are two primary use-cases for the `ignore` semantic:

1. reducing the run-time of trusted code (so we don't *need* to check)
2. escape-hatch for incorrect checks

(1) barely applies in the exception-specification case; implementations both gain better optimization opportunities if they actually enforce the check (so, the negative of what (1) is supposed to do), and have expended effort in making that check either free or almost-free.

(2), however, is not *that* far-fetched. Nannying users is not the C++ way, and if something falls out of a symmetry, we shouldn't disallow it with prejudice. We can recommend a warning, though.

4.1.5.1 Recommended practice

Implementations should warn if the `ignore` semantic is set on an `excspec` check.

4.2 Can we even do this?

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 if `std::terminate` happens to get called because of a bug - but it seems doubtful that someone would rely on an exception calling *specifically* `std::terminate()` instead of calling `std::terminate()` explicitly.

Any terminating semantic works just as well; and letting the violation handler be invoked allows for more unified recovery from unexpected conditions than the status quo.

Furthermore, it's no more an ABI break than the rest of the contracts facility; this paper has the same ABI implications as [P2900R5] does.

Therefore: if we can do contracts, we can do *this* paper.

4.3 On propagating exceptions out of `noexcept` functions

`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.

That said, allowing exceptions to propagate from `noexcept` functions is extremely dangerous due to algorithm choice, and the business function owner must do careful trade-off analysis to determine whether the greater danger is unwinding or termination. The flip side is - at present, they aren't permitted such an evaluation – the standard has made it for them.

Also note that this is **NOT THE PROPOSED DEFAULT**. The default is the status-quo.

4.3.1 Stack unwinding past `noexcept` and ABI

There are pretty obvious problems when linking `noexcept` functions between libraries compiled with the `ignore` or `observe` and some kind of forced-termination semantic. This paper acknowledges that, and recommends that implementations do not allow such foolishness, similarly to mixing `-fPIC` and non-`-fPIC` modes; or libraries compiled with a givensanitizer vs libraries that were not (some combinations work, others don't).

Ville Voutilainen suggested a separate resolution based on Bjarne's "nuclear plant use-case" [TODO CITE PAPER].

Consider the following case:

— `f()`, compiled with a non-terminating semantic (`ignore`/`observe`) calls `h()`, which `g() noexcept`, both compiled with a terminating semantic on `excspec` checks.
— `h()` has no unwinding tables for the scope where it calls `g()`, as `g()` cannot exit with an exception.

In this case, it might be better to specify that destructors for objects in `h()` do not get run, as a trade-off for allowing careful users to do best-effort nonterminating recovery.

If compiled with a build flag canceling `noexcept` optimizations, `h()` could still have an unwinding table entry, and all would be well. Effectively, users using potentially-throwing termination handlers get what they bargained for.

Note: the above is all QoI; the standard has no say in any cross-compilation unit shenanigans.

4.4 On Negative Testing

The "Lakos rule" – "do not mark narrow-contract functions `noexcept`" – exists for one reason, and it is testing defensive precondition checks in function implementations by throwing exceptions.

This is useful to a part of the C++ developer community. `noexcept` functions effectively make this technique impossible, even though the `noexcept` property is highly useful in more contexts than `move` and `destroy` operations.

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.

Most negative-testing scenarios can be better handled by a facility such as proposed in [@D3138R0].

4.4.1 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 failed precondition emits the violation object
      throw MyTestException(violation);
  }
  // 2 - if the exception-in-flight is the one we just threw in (1)
  try {
```

```
    std::rethrow_current_exception(); // do a Lippincott-switch [TODO check function name]
  } catch (MyTestException const&) { // it's a test exception
    throw; // rethrow it // 3
  } catch (...) {
    // for other exceptions, noexcept is still noexcept
    invoke_default_violation_handler(violation);
  }
}
```

Consider the function-under-test `sqrt`, with the above violation handler installed, and a test-driver below.

```
float sqrt(float x)
  noexcept // (f2)
  pre (x >= 0); // (f1)
```

Test-driver:

```
try {
  sqrt(-1f);
  // (f1) gets hit, and fails.
  // handle_contract_violation (1) throws MyTestException(violation)
  // MyTestException hits sqrt's noexcept boundary (f2)
  // handle_contract_violation rethrows the current exception (3)
  test_failure(); // never invoked, we're handling a MyTestException
} catch (MyTestException const& exc) { // we get here
  test_success(); // test succeeds
}
```

4.4.2 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.

The point is to enable making the trade-off, not allowing unsafe code. Failure modes are often at direct odds, and anwering "what is safe" requires the business context of an application.

4.5 Throwing while an exception is in-flight

Throwing an exception (say, from a destructor) during stack unwinding while an exception is in flight is also a terminating condition, and also clearly a form of erroneous behavior.

We suggest also treating this condition as a contract violation; the intricacies of this case will probably spin off into a second paper, however.

Note that before entering the violation handler, the exception is caught - a throwing violation handler under `observe` will not cause a parallel exception to propagate (and cause termination).

4.6 On `noexcept(contract_assert(false))`

If this paper is accepted, it would become very difficult to argue for the expression

```
// status quo: contract_assert(cond) is a statement
// this is not legal cod3
noexcept(contract_assert(false))
```

to be `false`.

With this paper, `noexcept` would come to mean *does not throw in-contract*; and the chosen semantic would *actually* deterimine whether the function can actually throw; the default, however, would not lie.

5 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.

6 Acknowledgements

- Jonas Persson contributed comments with regards to `noexcept` destructors, double throws, and unwinding code generation overhead.
- Ville Voutilainen contributed his usual bushel of insightful ruminations, the answers to which were worked into the paper.

7 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