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

Document #: D3205R0
Date: 2024-04-15
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>
Ben Craig
<ben.craig@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 Köppe's [P2795R5] 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.

In fact, looking at the list at 14.6.2 [except.terminate], it seems *all* the bullet points deal with some kind of erroneous behaviour. Each point is discussed below.

This paper brings breaching the `noexcept` specification of a function into the same discussion, along with all the other points where the language calls `std::terminate` and, since the Contracts facility aims to let users configure what to do when Erroneous behavior happens, allow novel ways of handling this eventuality.

The most delicate of these addtions of EB is classifying 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). It is delicate because it would allow an exception to be thrown if a throwing violation handler were installed. This paper argues this is no different than any other instance of notionally nonthrowing code throwing because of a contract violation.

We propose that `std::terminate` remains the "fallback behavior" in the case the contract is not enforced, which means the status-quo is conforming.

2 Proposed semantics

Note: in the current contracts design, a contract assertion has a *kind* (`pre`/`post`/`contract_assert`). Each assertion, separately, also has a *semantic* (see table below). The configurability of those is up to the implementation, but it seems reasonable that one dimension offered is per-kind.

2.1 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 pseudocode, the change we would like to make is

C++23	Proposed

`int halve(int x)`
`  interface {`
`    try {`
`      implementation;`
`    } catch (...) {`
 	
`      std::terminate();`
`    }`
`  }`
`{ return x/2; }`
	`int halve(int x)`
	`  interface {`
	`    try {`
	`      implementation;`
	`    } catch (...) {`
	`      contract_assert(false); // EB`
	`      std::terminate(); // fallback`
	`    }`
	`  }`
	`{ return x/2; }`

(The terms above are defined in the definitions section.)

The `contract_assert` above needs to have a different assertion kind, perhaps `implicit` (compare [P3100R0]).

Note: with `ignore` semantics, we get the status-quo behaviour; with a terminating semantic ("Louis" or `enforce`), we terminate according to that semantic instead.

2.2 A new assertion kind: `implicit`

If we are to treat Erroneous Behaviour as violations of implicit contract specifications, we need a new assertion kind, also proposed in [P3100R0].

2.3 Fallback Behavior

Erroneous Behavior is "well-defined" but Erroneous (and diagnosable). This implies the existence of a **Fallback Behavior** for each instance of Erroneous Behavior, which is whatever "well-defined" thing the implementation does if Erroneous Behavior occurs.

For the (proposed) Erroneous Behaviour this paper deals with, the Fallback Behavior should be the status-quo: `std::terminate()`.

If we treat Erroneous Behaviour as a contract violation, it therefore means that `std::terminate()` is called whenever such a contract check is not enforced.

2.4 Survey of contract semantics

(A discussion of this table is also in [P3237R0]).

semantic	checks	calls handler	assumed after	terminates	proposed
assume	no	no	yes	no	no
ignore	no	no	no	no	[P2900R5]
"Louis"	yes	no	yes	`trap`-ish	[P3191R0]
observe	yes	yes	no	no	[P2900R5]
enforce	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" [P3191R0] semantic (for Louis Dionne), in particular, has barely any reason to exist if it is not fixed at compile-time (its main use-case is reducing code-bloat).

The semantics that call the handler (`observe` and `enforce`) may throw; we have to deal with propagating that exception.

2.5 Proposed new Erroneous Behaviour

Along with the bullet points in 14.6.2 [except.terminate], `[[throws_nothing]]`, (deprecated) exception specifications, and Lewis Baker's [P3166R0] also work in this space, and should be treated similarly.

We give them names, to make them easier to discuss later.

2.5.1 Throw-before-landing

> (1.1) when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception ([except.throw]), calls a function that exits via an exception,

This is a *clear* erroneous situation. We are trying to invoke the landing pad but are interrupted by having to throw another exception. This should not happen in a correct program. (This scenario happens, for instance, if the exception is caught by value and its copy constructor throws).

2.5.2 Landing-pad-search failed

> (1.2) when the exception handling mechanism cannot find a handler for a thrown exception ([except.handle])

The **landing pad search** failed. That is clearly Erroneous Behaviour, the caller should have supplied a landing pad. It would be useful if it were possible for an implementation to diagnose this as EB.

2.5.3 `noexcept` (or-landing pad search cut off)

> (1.3) when the search for a handler encounters the outermost block of a function with a non-throwing exception specification

This suggests that `noexcept` is really a **landing pad search boundary**, and *not* "this function doesn't throw". In fact, argument construction happens outside the `noexcept` function boundary. Any call of a `noexcept` function that needs to construct arguments using non-`noexcept` code could still throw.

2.5.4 Throw while Stack-unwinding

> (1.4) when the destruction of an object during stack unwinding terminates by throwing an exception

This is also clearly EB - the issue is that if we are already unwinding the stack, then the exception will also unwind the stack, just more vigorously. Since this is clearly impossible, we choose to terminate.

2.5.5 Static-init

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

If you don't have a scope, how do you unwind? This should be diagnosable as EB.

2.5.6 Static-destroy

> (1.6) when destruction of an object with static or thread storage duration exits via an exception (6.9.3.4 [basic.start.term])

Again, impossible to unwind, especially when not in block scope. This should be EB.

2.5.7 Cleanup-callbacks

> (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])

We should require `noexcept` on these handlers, but that ship has sailed, so we should make it EB to throw from them.

2.5.8 Rethrow nothing

> (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])

Trying to throw something that isn't there is a runtime error. It's Erroneous. Let's make it so.

2.5.9 Rethrow wrapped nothing

> (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])

Same as before, this is clearly erroneous. This is a precondition violation - the precondition being "you're calling a function with one argument, you probably should supply it."

2.5.10 Thread-init

> (1.10) when execution of the initial function of a thread exits via an exception (33.4.3.3 [thread.thread.constr])

This is again a "we don't know how to unwind" scenario.

2.5.11 Parallalel-algorithms

> (1.11) for a parallel algorithm whose `ExecutionPolicy` specifies such behavior (22.12.4 [execpol.seq], 22.12.5 [execpol.par], 22.12.6 [execpol.parunseq]), when execution of an element access function (27.3.1 [algorithms.parallel.defns]) of the parallel algorithm exits via an exception (27.3.4 [algorithms.parallel.exceptions])

The problem here is that we don't know how to coalesce multiple exceptions from parallel branches. EB.

2.5.12 Joinable-thread

> (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 (33.4.3.4 [thread.thread.destr], 33.4.3.5 [thread.thread.assign])

Another instance of "This is just incorrect but it's better to specify what it does". It should be EB.

2.5.13 Wait-postconditions

> (1.13) when a call to a `wait()`, `wait_until()`, or `wait_for()` function on a condition variable (33.7.4 [thread.condition.condvar], 33.7.5 [thread.condition.condvarany]) fails to meet a postcondition.

It's even in the descriptions! This is a postcondition violation. It's EB.

2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 Source-location for such violations in `contract_violation` objects

- *If* the implementation chooses to make the semantic of `implicit` 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.

3 Discussion

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

3.1 Program flow under various semantics

… in which we discuss the behaviour of the proposal under various proposed semantics, for the `noexcept` case; the other cases are not ambiguous.

3.1.1 All the non-`noexcept` cases

In the non-noexcept cases, calling the violation handler first instead of `std::terminate` should be an ultimately implementable thing (we call one function now, we can call a different one too). The question arises mostly around what to do if a violation handler throws.

3.1.1.1 Nowhere to go: "Static-init", "Static-destroy", "Cleanup-callbacks", "Thread-init", "Joinable-thread"

We really just cannot do anything but terminate. Throwing here would just recurse into one of these cases again, so we must terminate the program if the handler *also* throws.

3.1.1.2 Precondition / Postcondition violations: "Wait-postconditions", "rethrow nothing", "rethrow wrapped nothing"

These are really just regular non-ignorable contract violations. Throwing from a handler can do normal exception propagation.

3.1.1.3 Exception Coalescing: "Throw while stack unwinding", "Parallel-algorithms"

We could either treat this situation as a "nowhere to go" case, or allow the exception thrown from the handler to *replace* any exceptions that are in flight, potentially allowing recovery by finding the application-level contract-violation handler as opposed to directly terminating.

This could very well leak the exceptions in flight; it still might be better than directly terminating.

Allowing the handler to replace the active exceptions would allow programmable exception coalescing, although we'd need some way to get at the uncaught exceptions to do that, and this paper makes no additional provision for that.

OPEN QUESTION: What does SG21 feel like? Any use-cases? Any implementation ideas?

3.1.1.4 Landing-pad search cases: "landing-pad-search failed", "`noexcept`", `[[throws_nothing]]`, other exception specification violations

This is the titular topic of the paper - this paper proposes that the exception that triggered the behaviour is *caught* outside the handler, available as `std::current_exception()`. This happens at the point where the exception failed to find a landing pad, or observed the EB - outside the function with the exception specification.

This means that throwing from the handler restarts the search for the landing pad; which means one can potentially find it.

It also means exceptions can escape `noexcept` functions if carefully managed; this is a feature, not a bug, but it *is* a very sharp knife. Fortunately, it seems to be the same sharp knife as throwing violation handlers in general.

3.2 Exception specification discussion

3.2.1 Default: the ignore semantic

Under the `ignore` semantic, we get the status-quo; the fallback behavior is invoked in all cases.

3.2.2 The "Louis" semantic

Under the "Louis" [P3191R0] (for Louis Dionne) 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.

3.2.3 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.

See "observe" for the notes on recovery through exceptions, however, since the handler does get to throw, which does not lead to termination.

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

3.2.4 The `observe` semantic

The `observe` semantic is very interesting for the same reason `enforce` is - 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.

3.2.4.1 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.

3.2.4.2 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 [P2698R0] - stack unwinding is sometimes still the best of bad options when we have erroneous behaviour, instead of termination.

NOTE: this is roughly on-par in danger 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.

3.2.4.3 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).

3.2.5 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.

Case 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.

Case 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.

3.3 Can we even do this?

The current meaning 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.

3.4 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.

The "no-throw" guarantee [Abrahams] is a property that developers rely on for program correctness. The "no-throw" guarantee isn't always used in conjunction with `noexcept`. In the standard, it is often spelled as "Throws: Nothing.". If a function that is documented as "Throws: Nothing" starts throwing, then program invariants can start to break. Exceptions escaping from `noexcept` functions is no worse than throwing from a "Throws: Nothing" function in terms of program correctness.

[P2946R1] also proposes a way to annotate functions with the "no-throw" guarantee, while allowing them to still throw. This is done with a configurable `[[throws_nothing]]` attribute, and it is very similar to this proposal, except that it uses a new facility rather than modifying the existing `noexcept` facilities, and that it doesn't (at present) interact with P2900's contract violation handlers.

Consider the following code example:

```
struct reservations {
    using resource_id = int;
    static constexpr size_t capacity = 2;

    size_t m_size = 0;
    resource_id m_reservations[capacity];

    resource_id operator[](size_t i) const pre(i < m_size && i < capacity) {
        return m_reservations[i];
    }

    void add_reservation(resource_id id) pre(id + 1 < capacity) {
        ::reserve_resource(id);
        m_reservations[m_size++] = id;
    }

    void clear()
    /* perhaps noexcept ? */
    /* perhaps [[ throws_nothing ]] ? */
    /* perhaps "Throws: Nothing." ? */
    {
        for (size_t i = 0; i < m_size; ++i) {
            ::release_resource((*this)[i]);
        }
        m_size = 0;
    }

    ~reservations() { clear(); }

    reservations &operator=(reservations &&other)
    /* perhaps noexcept ? */
    /* perhaps [[ throws_nothing ]] ? */
    /* perhaps "Throws: Nothing." ? */
    {
        if (&other == this)
            return *this;
        clear();
        for (size_t i = 0; i < other.m_size; ++i) {
            m_reservations[i] = other[i]; // violation when i >= capacity
            ++m_size;
        }
        other.m_size = 0;
        return *this;
    }
};

void race(reservations &r) {
    // buggy code
    std::jthread t1([&] {
        r.add_reservation(g_resource1);
        // ...
    });
    std::jthread t2([&] {
        r.add_reservation(g_resource2);
        // ...
    });
}

void fragile_clear() {
    reservations r1;
    r1.add_reservation(g_resource0);
    race(r1);
    r1.clear(); // BOOM

    // clear() and  ~reservations both release (at least) g_resource
}

void fragile_move() {
    reservations r2;
    r2.add_reservation(g_resource0);
    race(r2);
    reservations r3;
    r3 = std::move(r2); // BOOM

    // r2 and r3 both release (at least) g_resource0
}
```

The contract on `operator[]` is exactly the kind of high value contract that those concerned about safety and security are most interested in. Developers expect `operator[]` to have the "no-throw" guarantee. Developers also expect to be able to get performance benefits by marking move operations as `noexcept`. There is no clear right answer as to how to combine the "no-throw" guarantee with throwing contract violation handlers.

Strategy	fast std::vector resize	library UB	terminates
"Throws: Nothing."	no	yes	no
terminate `[[throws_nothing]]`	no	no	yes
throwing `[[throws_nothing]]`	no	yes	no
terminate `noexcept`	yes	no	yes
throwing `noexcept`	yes	yes	no

With a configurable `noexcept`, the end developer (rather than the library developer) can determine whether the greater danger is unwinding or termination, all while still getting the benefits of better algorithm selection. The flip side is - at present, they aren't permitted such an evaluation – the standard has made it for them.

Also note that allowing exceptions to escape `noexcept` functions is **NOT THE PROPOSED DEFAULT**. The default is the status-quo.

3.4.1 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 given sanitizer 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" [P2698R0].

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 `noexcept` checks.
— `h()` has no unwinding tables for the scope where it calls `g()`, as `g()` cannot exit with an exception.

We need to do some research to see if we can specify a "you get what you get" for unwinding `h()`'s frame on every ABI - on some, the "worst" that would happen is missing some destructors, but on others, it might not be implementable (perhaps the unwinder cannot locate the stack pointer).

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.

3.5 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 [P3183R0], which allows testing `pre` and `post`-assertions without invoking the function.

The "Lakos rule" can be seen as a "good" solution, throwing violation handlers for `noexcept` csn be seen as a "better" solution, and [P3183R0] assertion execution can be seen as the "best" solution.

3.5.1 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)
  // we are already in some kind of catch (...) block (see Semantics)
  auto excptr = std::current_exception();
  try {
    // do a Lippincott-switch
    std::rethrow_exception(excptr);
  } 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
}
```

3.5.2 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.

3.6 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 code
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.

4 Definitions

This secition defines the terms **prologue** and **epilogue**; it shows how `pre` and `post` assertions map to them, and how the exceptions in the implementation are handled. **It does not propose anything**

As per Lisa Lippincott’s work on function interfaces [P0465R0], 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’s **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’s **epilogue** (POST2, T2, T3, T4) notionally runs after the return value has been constructed (in non-exceptional cases) 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 (POST1, POST2) are equivalent to [P3098R0]'s postcondition captures `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 [P2946R1] `[[throws_nothing]]` (if the `contract_assert` has the `ignore` semantic), and would be equivalent to `noexcept` if it were reflectable (with the `noexcept()` query). This paper provisionally labels this section with the `implicit` assertion kind.

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.
— Bengt Gustaffson contributed a wonderfully thorough review and suggested further possible ABI issues.

7 References

[Abrahams] David Abrahams. 2001. Lessons Learned from Specifying Exception-Safety for the C++ Standard Library.
https://www.boost.org/community/exception_safety.html

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

[P0465R0] Lisa Lippincott. 2016-10-16. Procedural Function Interfaces.
https://wg21.link/p0465r0

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

[P2698R0] Bjarne Stroustrup. 2022-11-18. Unconditional termination is a serious problem.
https://wg21.link/p2698r0

[P2795R5] Thomas Köppe. 2024-03-22. Erroneous behaviour for uninitialized reads.
https://isocpp.org/files/papers/P2795R5.html

[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. A flexible solution to the problems of `noexcept`.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2946r1.pdf

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

[P3098R0] Joshua Berne. 2024. Contracts for C++: Postcondition captures.
https://isocpp.org/files/papers/P3098R0.html

[P3100R0] Timur Doumler. 2024-04-15. Contracts, Undefined Behaviour, and Erroneous Behaviour.
https://isocpp.org/files/papers/P3100R0.html

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

[P3166R0] Lewis Baker. 2024. Static Exception Specifications.
https://isocpp.org/files/papers/P3166R0.html

[P3183R0] Bengt Gustaffson. 2024-04-15. Contract testing support.
https://isocpp.org/files/papers/P3183R0.pdf

[P3191R0] Louis Dionne. 2024. Feedback on the scalability of contract violation handlers in P2900.
https://isocpp.org/files/papers/P3191R0.pdf

[P3237R0] Andrei Zissu. 2024-04-16. Matrix Representation of Contract Semantics.
https://isocpp.org/files/papers/P3237R0.pdf