Throwing Violation Handlers Are Post Hoc Library Design

Document Number: P4254R0
Date: 2025-06-02
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

Abstract

This paper considers the Lakos Rule [1] and how it negatively interacts with the process of library design.

Background

The Lakos Rule [1] has long been treated as a trump card in LEWG: A proposal contains a function with a precondition which is declared `noexcept`, someone in the room invokes “the Lakos Rule,” and the function must presumptively be redesigned to remove the `noexcept`. This has impeded the adoption of `noexcept` in the library, and has led to arguably-ridiculous consequences (e.g. `std::vector::operator[]` not being marked `noexcept` despite being distinguished from `std::vector::at` only by the fact that it does not throw).

Despite the above the Lakos Rule is not actually standing LEWG policy [2].

C++26 brings contracts, and therewith the “violation handler” which is permitted to throw [3]. This allows precondition violations to be transformed into an exception at a global level. Such violation handlers may be referred to as “throwing violation handlers.” I voted strongly against the adoption of P2900 in LEWG in Wrocław due to throwing violation handlers.

C++26 also brings “hardened preconditions” to the library [4]. In a “hardened implementation” violations of hardened preconditions trigger a C++26 contract violation (which may, due to throwing violation handlers, be realized as an exception). I voted strongly against the adoption of P3471 in LEWG in Hagenberg due to the interaction of hardened preconditions and throwing violation handlers giving the following rationale:

"LEWG has repeatedly deferred considering noexcept vis-a-vis preconditions and now we’re shipping the conclusion of that discussion in 30 minutes as an indirect consequence of another paper."

It is now being argued [5] that the above renders the Lakos Rule *fait accompli* (exactly as I articulated in my justification for my vote on P3471 in Hagenberg).

Discussion

Preconditions and Library Design

Library design is a considered process. The problem domain is analyzed and subdivided and a library emerges therefrom, representing a certain decomposition of the problem and realization of the solution. One of the tools that can be used in this process is a precondition.

A precondition is a contract between caller (who provides the precondition) and callee (who assumes it). It describes the state of the universe which must be true in order for a certain function call to occur.

For example: `std::vector::operator[]` has as a precondition that the sole argument thereto will be “in range” (i.e. will indicate an index which has an element).

Preconditions are an important part of library design in that they inform the design of the function to which they are attached. Continuing with the above example the design of `std::vector::operator[]` does not include a way to report an error, because it has defined the possibility of such an error out of existence via its precondition. On the other hand the design of `std::vector::at` does include a way to report an error, because it does not have a precondition which excludes invalid inputs.

C++ functions have at least one, and at most two, means by which they can return control to their caller. The first and most obvious is regular function return, which can be taken to semantically indicate success (note that `[[noreturn]]` functions can only be definitively said not to return if you treat attributes as non-ignorable, which they aren’t). The second is by throwing an exception, which can be taken to semantically indicate failure.

The C++ language includes a means by which the above-described failure modality can be declared non-existent (ergo “one, and at most two” from above): `noexcept`. Functions marked `noexcept` cannot emit exceptions, which gives syntactic force to the semantics ascribed to said functions by their design.

Most C++ programmers likely agree that the type returned by a function should match its design, not some hypothetical design which wasn’t chosen. The Lakos Rule asks us not to apply this calculus to throwing-ness, instead syntactically indicating that functions we’ve explicitly said throw nothing (i.e. “Throws: Nothing”) are nonetheless capable of emitting exceptions.

This raises the question of what could justify the Lakos Rule. N3248 [1] articulates that justification thusly:

"When validating [...] defensive checks from a test driver, a reasonable approach is to register an assert-handler that throws a well-defined precondition-violated exception, which the test driver catches to ensure that the appropriate asserts are indeed in place."

Put differently: The justification for the Lakos Rule is to give precondition violations semantics. But this is exactly what the designer neglected to do when designing the API, it is the precise reason for which they chose to employ a precondition (for support of this consider that it is not an oversight that both `std::vector::operator[]` and `::at` exist).

If certain situations, proscribed by preconditions, ought to have semantics then this consideration should be undertaken when the library is designed. It should not be something that can be bolted on post hoc through installation of a throwing violation handler.

The above justification for the Lakos Rule centers around a user calling a function out of contract and complaining about the behavior they obtain. But the entire point of having a contract is exactly so users can't complain about the behavior they get when they violate them. When LEWG puts a precondition on a function are users allowed to call it out of contract or not?

Is This Code Correct?

Consider this function:

```
int foo(std::mutex& m, const std::vector<int>& v, const std::size_t s) {
  m.lock();
  const int i = v[s];
  m.unlock();
  return i;
}
```

Given it has the precondition that `s` must be a valid index into `v`, does it ever leak the lock it acquires against `m`?

Straightforward analysis indicates that it does not leak said lock, because `std::vector::operator[]` does not throw exceptions. The Lakos Rule, however, would have you believe that it is possible for this function to leak a lock against `m` when it is called out of contract (since the Lakos Rule model renders that a situation which warrants consideration).

A response to the above might be to posit that it doesn't matter whether a lock against `m` is leaked because in order for that to occur the function has to be called out of contract. This argument is exactly right, but it dovetails with the argument raised by the preceding section: When a precondition is violated, no claims are made as to the behavior of the function. As demonstrated by this example, however, the Lakos Rule engages in special pleading arguing

that no claims are made save that exceptions must be propagated. The program can be completely corrupted and meaningless as long as exceptions are allowed to flow.

Generic Code

The standard library supplies `std::invoke_result_t` which allows generic code to determine the type an invocable yields when invoked with a certain cv- and ref-qualification, and with arguments of certain types. Put differently the standard library provides first-class support for interacting with the value channel of generic invocables.

The standard library also supplies `std::is_nothrow_invocable_v` which allows for the same interrogation, save for the error channel (i.e. exceptions). No one thinks the result of `std::invoke_result_t` should reflect anything other than the actual behavior of the invocable. The Lakos Rule, however, would have `std::is_nothrow_invocable_v` yield `false` for invocables which everyone knows don't throw, just because they have a precondition (i.e. the template variable would claim an error channel exists when, in practice, it does not).

`std::execution`

Contracts was a major language feature which landed in C++26. A major library feature which landed in C++26 was `std::execution` [6]. `std::execution` brings best-in-class asynchronous programming support to C++ via a model referred to as "sender/receiver."

"Sender/Receiver itself is the implementation of an `async-function`. An `async-function` can complete asynchronously with values, errors, or cancellation." [7]

P2849 is not alone in drawing an analogy between `std::execution` and regular, synchronous functions [8][9].

The two completion channels of regular, synchronous functions are wildly different:

- Regular return is limited to a single type, and is usually taken to be inexpensive to employ, whereas
- Exception throw can transmit any type (even types which don't derive from `std::exception`), and is usually taken to be expensive to employ (which is why C++ developers often transmit error via the "value" channel or via an out parameter)

This is not the case for the asynchronous operations of `std::execution`. Two of the channels (value and error) have unbounded arity (i.e. an asynchronous operation may complete successfully or in error in any number of ways, including zero). All of the channels are equally expensive to transmit (being transmitted via a synchronous function call).

As with regular functions, asynchronous operations in `std::execution` declare the ways in which they can complete. This is done by instantiating

`std::execution::completion_signatures` with function types indicating each completion modality which will be employed by the associated asynchronous operation.

Note that the API surface of `std::execution` flagrantly flaunts the Lakos Rule:

- `std::execution::start` requires that the `start` member function of the operation state be marked `noexcept` despite having a precondition (i.e. `start` may only be invoked a single time on a certain operation state)
- `std::execution::set_value`, `::set_error`, and `::set_stopped` require that the `set_value`, `set_error`, and `set_stopped` (respectively) member functions of the receiver be marked `noexcept` despite having a precondition (i.e. only a single one of them may be invoked a single time on a certain receiver)

The reasons for this are clear and obvious to those that understand the design of `std::execution`: There is no reasonable way for a caller to react to exceptions emitted by the above-mentioned APIs (note that this was among the rationales for splitting `submit` into `connect` and `start` [10]).

The above calculus applies equally to `std::mutex::unlock`. That API, however, is designed in accordance with the Lakos Rule and since it has a precondition ("[t]he calling thread owns the mutex" §32.6.4.2.1 [thread.mutex.requirements.mutex.general]) it is not marked `noexcept`. The program syntactically claims the function has an error channel even though it either:

- Does not, in practice, have one, or
- Nothing useful can be done with any information ever communicated by that channel

The fact that `std::execution` is not designed in accordance with the Lakos Rule is not the only interesting thing here. Because `std::execution` constitutes a fundamentally-asynchronous ecosystem it realizes synchronous exceptions thrown during an operation as `std::execution::set_error(rcvr, std::current_exception())` (§33.9.1 [exec.snd.general]). This means that if any synchronous part of an asynchronous operation can throw an exception the overall asynchronous operation must advertise a completion signature of `std::execution::set_error_t(std::exception_ptr)` (§33.3 [exec.async.ops]). This means that failure to apply `noexcept` where it morally belongs (i.e. where the function is known not to throw when called in contract) results in:

- An additional completion signature being advertised (which results in more code being emitted and therefore in longer compilation times and larger object files),
- Additional storage possibly being allocated (for operations which need to store intermediate completions, such as `std::execution::when_all`), and
- The fundamental library design problem of an operation advertising a fundamentally-incorrect set of completion signatures

LEWG appears to be sympathetic to the above, having forwarded at least one paper whose only purpose is removing `std::execution::set_error_t(std::exception_ptr)` from the set of completion signatures when appropriate [11].

Consistent application of the Lakos Rule, however, pessimizes the above. Synchronous parts of asynchronous operations should, under the Lakos Rule, not mark themselves as `noexcept` if they have a precondition, thereby adding a `std::execution::set_error_t(std::exception_ptr)` completion where one has no reason to exist.

`std::execution` raises still more questions vis-à-vis the Lakos Rule, however. For example would a logical consequence of adopting/upholding the Lakos Rule be requiring asynchronous operations which have a precondition to advertise `std::execution::set_error_t(std::exception_ptr)` even if the operation's execution when connected and started in contract has no path which could possibly generate that completion? This would after all be analogous to the Lakos Rule's application to regular, synchronous code.

Lastly, what happens when the Lakos Rule interacts with asynchronous destruction/clean up [12], which is intuitively understood to be infallible (consider that destructors are `noexcept` unless explicitly declared `noexcept(false)`)? Do we need a parallel version of generic, asynchronous algorithms which don't advertise `std::execution::set_error_t(std::exception_ptr)` despite preconditions, for use only in asynchronous clean up? Do we need an adaptor which removes `std::execution::set_error_t(std::exception_ptr)` completions by transforming them to a call to `std::terminate`? Or should clean up primitives advertise `std::execution::set_error_t(std::exception_ptr)` with the understanding (which cannot be communicated between generic components) that it will never be used?

Death Tests

Program termination is not a general-purpose error handling mechanism [13]. If, however, a program encounters a condition which cannot be recovered from program termination is the fastest way to leave the unrecoverable program state [14]. The C++ programming language uses program termination to handle at least the following errors:

- A failed `assert`
- Allowing an exception to propagate out of a `noexcept` function
- Allowing an exception to propagate out of a `std::thread`
- Allowing an exception to propagate out of a destructor during stack unwinding

Library implementers can easily terminate on contract violation (P2900 provides first-class support for this) and detect a terminating subprocess if they want to test the behavior of functions called out of contract (i.e. a "death test").

Proponents of the Lakos Rule, however, reject this [15]:

“When targeting Windows, embedded platforms, or the browser, this approach either does not scale due to a much higher runtime overhead or is outright impossible due to lack of multiprocess support [...]”

This trifurcates the problem space:

- For Unix-like platforms death tests are workable, and therefore the Lakos Rule addresses only more minor concerns on those platforms (P2831 complains, for example, that `std::abort` does not carry as much semantic information as an exception)
- For platforms such as Windows death tests are workable in principle, but performance means one may wish to avoid them
- For platforms without multiprocess support death tests are unworkable

But recall the situation: These are not complaints about normal use of the standard library. They are not complaints about small details of the specification thereof. They are complaints about what happens when the standard library is used in a manner that is explicitly precluded by its specification.

The Lakos Rule posits that out of contract function calls are worth considering, contrary to the meaning of a precondition. Once that discussion is engaged with the Lakos Rule additionally requires us not only to accede to the idea that out of contract function calls should have semantics ascribable post hoc, but to also accept that these post hoc ascribable semantics ought to be maximally ergonomic for those making the out of contract function calls even at the expense of the API which is presented to users making calls actually allowed by the contract written into the standard.

Conclusion

The Lakos Rule asks that the entire standard library not syntactically encode what it means, for the benefit of those who want to call functions out of contract (which is proscribed by the preconditions attached to those functions) and observe the result thereof (which is intentionally not defined or guaranteed). It renders facially-correct code suspect by introducing throwing paths where the designs of the functions being composed say they oughtn’t exist. LEWG should reject the Lakos Rule and write the equivalent in code (`noexcept`) to what is said in prose (“Throws: Nothing”).

Proposal

“Throws: Nothing” should mean/imply `noexcept` [16].

Acknowledgements

The author would like to thank Eddie Nolan for reviewing this paper.

References

[1] A. Meredith et al. noexcept Prevents Library Validation N3248
[2] B. Lelbach et al. Library Evolution Leadership's Understanding of the noexcept Policy History P2920R0
[3] J. Berne et al. Contracts for C++ P2900R14
[4] K. Varlamov et al. Standard library hardening P3471R4
[5] noexcept policy for SD-9 (The Lakos Rule) P3155R0
[6] M. Dominiak et al. std::execution P2300R10
[7] K. Shoop. async-object - aka async-RAII P2849R0
[8] R. Leahy. Of Operation States and Their Lifetimes P3373R4
[9] I. Petersen. Towards Senders in Interfaces P4223R0
[10] L. Baker et al. Eliminating heap-allocations in sender/receiver with connect()/start() P2006R1
[11] R. Leahy. When Do You Know connect Doesn't Throw? P3388R3
[12] R. Leahy. It's Scopes All the Way Down P3955R0
[13] R. Leahy. Failing Successfully: Reporting and Handling Errors CppCon 2021
[14] R. Leahy. user_data Is a Return Address Or: I'm Not Unreasonable for Expecting Functions to Return C++Now 2026
[15] T. Doumler et al. Functions having a narrow contract should not be noexcept P2831R0
[16] B. Craig. noexcept policy for SD-9 (throws nothing) P3085R3