Functions having a narrow contract should not be `noexcept`

Timur Doumler (papers@timur.audio)
Ed Catmur (ed@catmur.uk)

Document #: P2831R0
Date: 2023-05-15
Project: Programming Language C++
Audience: Library Evolution Working Group

Abstract

The Lakos Rule is a long-standing design principle in the C++ Standard Library. It stipulates that a function having a narrow contract should not be declared `noexcept`, even if it is known to not throw when called with valid input. In this paper, we demonstrate why the Lakos Rule is still useful and important today and should not be removed.

1 Introduction

C++ functions — in the C++ Standard Library or in other places — can have *preconditions*, which are a form of *contract*. A function that has no preconditions on its input (parameter) values or on the state (object state or global state) accessible from it — i.e., a function that has defined behaviour for any combination of input values and accessible state — is said to have a *wide contract*. Examples of such functions in the C++ Standard Library are `std::vector::at` and `std::vector::size`.

If such a function is required to never throw an exception (or if it is somehow known that it will never throw an exception), it may be declared `noexcept` (conditionally or unconditionally). This is the case for `std::vector::size`.

By contrast, a function that has preconditions — i.e., a function whose behaviour is undefined[1] for some combination of input values and accessible state, which we can call *invalid* — is said to have a *narrow contract*. Examples of such functions in the C++ Standard Library are `std::vector::operator[]` and `std::vector::front`. Invoking the former with an out-of-bounds index or invoking either function on an empty vector will result in undefined behaviour.

A long-standing design principle in the C++ Standard Library has been that a function having a narrow contract should not be declared `noexcept`, even if it is known to never throw an exception for a *valid* combination of input values and accessible state. When a function having a narrow contract is obliged to not throw, the function should nevertheless *not* be declared `noexcept` but merely specified as *Throws: nothing*. This design principle allows for highly effective testing strategies that involve throwing exceptions as a way of diagnosing *contract violations* — i.e., bugs introduced by

[1]It is sometimes useful to distinguish between *library undefined behaviour* or *soft UB* (violating the preconditions of a function), which might be recoverable if the violation is detected at the time of the call, and *language undefined behaviour* or *hard UB* (hitting core undefined behaviour — see [P1705R1] — inside the implementation of the function), which is unrecoverable, although the C++ Standard itself does not make such a distinction.

calling the function with an invalid combination of input values and accessible state (calling the function *out of contract*). This design principle is also known as the *Lakos Rule*.

The Lakos Rule was first proposed in [N3248] and adopted with [N3279]. An updated version of the rule was codified into policy in [P0884R0]. See [O'Dwyer2018] for a more detailed summary.

More recently, [P1656R2] argued that the Lakos Rule should be abandoned as a design principle. According to this paper, functions that are known to never throw an exception for a valid combination of input values and accessible state should always be declared `noexcept`, regardless of whether they have a wide or a narrow contract. Further, [P2148R0] proposed adopting a new standing document with design guidelines for the evolution of the C++ Standard Library that move away from the Lakos Rule.

This paper makes the case that the Lakos Rule is still useful and important today and must be retained as a design principle for the C++ Standard Library. In section 2, we compare the various known techniques for negative testing, demonstrating that the Lakos Rule is essential for implementing negative testing effectively. In section 3, we present case studies from real-world codebases where the Lakos Rule is central to maintaining an effective testing strategy. In section 4, we argue why the Lakos Rule is not only important in such third-party codebases, but also for the C++ Standard Library itself. In section 5, we discuss why the urge to excessively use `noexcept` — often the reason why C++ developers do not follow the Lakos Rule — is misguided and what the actual use case for `noexcept` is. Finally, in section 6, we consider recent developments for standardising a C++ Contracts facility and discuss why the Lakos Rule is still needed if we have such a facility.

2 Negative testing

Unit tests are an established engineering practice to ensure software quality and a crucial part of the software test pyramid. Let us consider how we would unit test a function having a narrow contract, such as `std::vector::front`.

Writing unit tests for cases in which `front` is being called in contract and therefore has defined behaviour is straightforward. We establish valid combinations of input values and accessible state and test whether the function gives the expected output in each case:

```
std::vector<int> v = {1};
REQUIRE(v.front() == 1);
// etc.
```

Here, `REQUIRE` is some macro provided by the unit test framework to verify that the given predicate evaluates to `true`, report success or failure, and continue the execution of the test suite.

Now, what happens if we call `front` out of contract, i.e., on an empty vector? In this case, the behaviour is undefined. Calling `front` on an empty vector is therefore unconditionally a bug. This specification is necessary to achieve maximum performance, e.g., in a release build, where we cannot afford to check the precondition at run time. In a debug build, however, such a precondition check is possible and is in fact critically important to prevent the introduction of such bugs.

Since C++23 lacks a language-level Contracts facility (see section 6), we need to use a library-based solution to write the precondition check. Typically, this check will be implemented with some kind of assertion macro at the beginning of the function body:

```
T& front() {
  ASSERT(!empty());
  // implementation
}
```

Precondition checks are code and, just like any other code, ought to be tested. We therefore need to write a unit test to ensure that the precondition check has in fact been added. This kind of testing is sometimes called *negative testing*:

```
std::vector<int> v;  // empty
REQUIRE_ASSERT_FAIL(v.front());
```

Negative testing is critically important: without a negative test, we cannot be sure that the developer of the `front` function considered this case and added a check that will alert users of `front` about out-of-contract calls and prevent them from introducing bugs. But how do we write such a negative test? How do we implement `REQUIRE_ASSERT_FAIL` in our testing framework?

Once we hit the `ASSERT` macro and the contract check fails, continuing to execute the body of the function is no longer meaningful; the code will either crash or exhibit some other form of undefined and potentially harmful behaviour. To continue running our unit test suite, we therefore need a way to exit the function — other than by returning a value — at the point where the contract violation occurred and communicate detailed information about the contract violation back to the testing framework. Below we discuss the known strategies to achieve such a controlled function exit.

2.1 Exception based

The most natural, portable, and effective way to exit the function without continuing to execute the function body (which would invoke undefined behaviour) is to throw an exception at the point where the contract violation occurred. We can define our `ASSERT` macro as follows[2]:

```
#if TEST_ASSERTIONS
  #define ASSERT(expr) if (!expr) throw AssertFail();
#else
  // other possible actions: ignore, assume, log and continue, log and terminate
#endif
```

Then, in `TEST_ASSERTIONS` mode (which will often, but not always, correspond to debug mode), we can define our `REQUIRE_ASSERT_FAIL` to verify that an exception of type `AssertFail` has been thrown, report success or failure, and continue the execution of the test suite. This is efficient, portable, and straightforward: every modern C++ testing framework provides a way to check for a thrown exception of a particular type, and if necessary it is easy to write such a check by hand.

Another important advantage of exception-based negative testing is that we can communicate an arbitrary amount of information about the contract violation back to the testing framework via the thrown exception object. [P1656R2] repeats the canard that stack unwinding destroys information. This claim might be true in the naïve case, but any sophisticated implementation will collect the relevant information before stack unwinding, either immediately before throwing the exception or (for more general benefit) at the end of the search phase, under the control of the catch block but before stack unwinding begins.

The only issue with exception-based negative testing as described above is that it no longer works if the function under test is declared `noexcept`. Throwing an `AssertFail` out of a `noexcept` function would immediately result in `std::terminate`, bringing down the whole test suite.

2.1.1 Following the Lakos Rule

The obvious way to solve the `noexcept` problem is to not declare a function with a narrow contract `noexcept`, even if we know that the function will never throw when called in contract. In other words, exception-based negative testing is straightforward if we just follow the Lakos Rule.

[2] At Cradle, we have a slightly more sophisticated definition: when debugging locally, i.e., if a debugger is attached, the `ASSERT` macro will trigger a breakpoint on contract violation, using utilities like the ones proposed in [P2514R0]; otherwise (that is, when running the test suite on CI or locally but without a debugger attached), it will throw an `AssertFail` exception as shown here.

If the Standards committee abandons the Lakos Rule as a design principle (as proposed in [P1656R2] and [P2148R0]), functions such as `std::vector::front` might be specified as `noexcept` in a future standard. This new direction would make writing negative tests (and, therefore, preventing bugs from being introduced because of out-of-contract calls and missing contract checks) much harder. In the remainder of this section, we discuss various workarounds and their shortcomings compared to the straightforward exception-based technique that the Lakos Rule enables.

2.1.2 Conditional `noexcept` macro

A workaround used by some libraries is to introduce a macro along the lines of

```
#if TEST_ASSERTIONS
  #define MY_NOEXCEPT
#else
  #define MY_NOEXCEPT noexcept
#endif
```

Then, we can annotate all functions having a narrow contract with `MY_NOEXCEPT` instead of `noexcept` proper. Thus functions having a narrow contract can be `noexcept` in production, and at the same time, we can use exception-based negative testing on them when compiled in `TEST_ASSERTIONS` mode.

This option, however, is unsatisfactory because we effectively end up unit testing not our actual code but code compiled with a different specification, which may result in different behaviour: switching the `noexcept` specification of a function depending on the build mode can trigger different code paths being taken. This is observable by users (for example, turning moves into copies) and causes confusion. Software engineering best practice fairly demands that we test the actual code that is built for production, which is not possible with this technique. That is why libc++ ultimately decided against this approach after having introduced it; see 4.1. See also [P2834R0] which explains from first principles why this approach is such a bad idea.

2.2 `setjmp` and `longjmp`

Another way to exit the function from our `ASSERT` macro on contract violation is to use `setjmp` and `longjmp`. However, this technique does not work for negative testing. With most compilers,[3] when using `setjmp` and `longjmp` instead of `throw` and `catch`, the stack is not unwound and destructors of objects on the stack are not called. The C++ Standard specifies in [csetjmp.syn]:

> The contents of the header `<csetjmp>` are the same as the C standard library header `<setjmp.h>`.
>
> The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this document. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any nontrivial destructors for any objects with automatic storage duration.

The specification above means that in practice, we will immediately run into undefined behaviour when performing negative testing of any C++ code involving objects having nontrivial destructors. Most real-world C++ code calls such destructors. But even if the behaviour were defined, if we run thousands of unit tests involving data structures that allocate significant amounts of memory on the heap, we end up with an unacceptable number of memory leaks (and memory usage is often an integral part of thorough unit testing). We also break the program logic in the presence of other resources that rely on RAII, such as `std::lock_guard`. For all these reasons, this approach is not viable.

[3]Notably, Microsoft's implementation of `setjmp` and `longjmp` *does* perform stack unwinding with local object destruction, as is done for `throw` and `catch` (see [MSVCDocLongjmp]), while GCC and Clang do not.

2.3 Child threads

Another strategy for negative tests that does not involve throwing exceptions or performing death tests is to invoke the function under test in a child thread. On contract violation, the `ASSERT` macro can save some information about the violation and then lock the thread (by putting it to sleep indefinitely, or perhaps spinning in an infinite loop). `REQUIRE_ASSERT_FAIL` can then verify that this has happened.

This approach is slightly more comprehensible than `setjmp` and `longjmp` and does not suffer from the undefined behaviour issue but still has all the other drawbacks of `setjmp` and `longjmp`, such as leaking memory (invalidating unit tests that track such leakage) and breaking any program logic relying on RAII. This approach also leaks one thread for every test case.

2.4 Stackful coroutines

Ville Voutilainen recently suggested yet another approach for negative testing without throwing exceptions or performing death tests. While this approach is in many ways a thought experiment, and we are not aware of any testing framework or codebase successfully using this approach, it has been successfully prototyped[4].

The idea is that on contract violation, the `ASSERT` macro would yield to a cooperative scheduler. The tests would have to be written in such a way that the run of a subsequent test would be triggered by the event loop in conjunction with triggering the previous test's result verification. A failed test will run a nested event loop (which is the scheduler yield) and get stuck there, without proceeding to run the body of the function called out of contract. A successful test will just run its code and then call the nested event loop. In both the failed and the successful cases, the event loop will subsequently run the test result verification of the test, and then the event loop will run the next test. The sequence of tests thus becomes a sequence of recursive calls, and each failed test behaves effectively like a suspended stackful coroutine.

Just like the `setjmp`/`longjmp` and child thread approaches, this approach would never call destructors of any parameters or objects created by the test call, therefore leaking memory (invalidating unit tests that track such leakage) and breaking any program logic relying on RAII. In addition, the call stack would keep growing with every test call, consuming a large and unbounded amount of stack space. Finally, the whole test suite would have to be arranged in a very particular way in order to make this technique work, and would require a testing framework that offers the required event loop machinery.

2.5 Signals

Signals have been suggested as another way to exit the function from our `ASSERT` macro. However, signals do not help us here either. First of all, although synchronous signals are available on POSIX platforms, they are not available on Windows and are therefore not viable for cross-platform development. More importantly, if, on contract violation, we raise a signal in `ASSERT` and then install a custom signal handler to handle it, we can do only one of two things at the end of such a signal handler: either return control back to the function that raised the signal, or terminate the program. Using signals is therefore no different from using any other callback-based approach (see above).

2.6 Death tests

If we cannot continue executing the body of the function under test but have no practical way to exit the function other than by terminating the entire process, the only remaining option for

[4]For a prototype implementation in standard C++ that can be experimented with on Compiler Explorer, see `https://godbolt.org/z/obsfvzrqh`.

negative testing is to implement it as a *death test*. In a death test, the code under test is run in a separate process. A contract violation in the `ASSERT` macro leads to termination of this process with some error message. `REQUIRE_ASSERT_FAIL` verifies that the process has been terminated and that an error message has been triggered. In principle, this approach works, but several drawbacks make it a nonviable solution for many codebases.

We are aware of three ways to implement death tests: fork based, clone based, and spawn based.

2.6.1 Fork based

In a fork-based death test, each negative test is run in a forked process. This kind of test works reasonably well on platforms having a fast, reliable `fork()`. In practice, use of this fork-based approach limits us to UNIX-like platforms, such as Linux and macOS. Fork-based death tests can therefore be a viable strategy if your C++ library targets only these platforms.

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 is a major reason why most C++ unit test frameworks do not support death tests. From the five most popular C++ unit test frameworks, only GoogleTest supports death tests, while Catch2, Boost.Test, CppTest, and DocTest do not.

Another drawback is that even on platforms where death tests can be implemented efficiently, they can carry only a small amount of information about the contract violation; by using `std::_Exit` instead of `std::abort`, one can communicate up to 8 bits of information. This amount of diagnostic information is very meagre compared to the unlimited amount of information (such as the source location and, in advanced usage, the values of operands) available to be carried on an exception from a failed assert handler. Some more information can be carried through standard streams, but this approach is fragile and requires the rigmarole of serialisation and deserialisation.

2.6.2 Clone based

On Linux, `clone()` can be used instead of `fork()`. This approach has the advantage that `clone()` is less likely than `fork` to cause the child to hang when the parent process has multiple threads (see [GTestDocDeathTests]). However, `clone()` is even less portable than a fork-based death test, since it works only on Linux.

2.6.3 Spawn based

A different flavour of death test that does not depend on `fork()` or `clone()` is a spawn-based death test, where the testing framework spawns a new process for each negative test. But spawn-based death tests have several drawbacks compared to fork-based and clone-based death tests: typically, they require adoption of an external, usually non-C++, testing framework (DejaGNU, lit, CTest, make); they require moving test code into other source files, making it more difficult to track; and they require building the state for each test from scratch. On the other hand, fork-based death tests (and exception-based negative tests) can build up and reuse state. All this makes spawn-based negative tests orders of magnitude more cumbersome to write and the adoption of such tests much less likely, leading to worse software quality.

Like fork-based death tests on non-UNIX-like platforms, spawn-based death tests also suffer from a very high performance overhead. A mid-sized test suite may have several thousand negative tests. The overhead of spawning that many processes, even on platforms where that is relatively fast, is enough to turn a test suite that runs in under a second into one that takes minutes. That performance degradation alone precludes test-on-save, red-green-refactor, and other modern development processes.

3 Case studies

A well-known codebase that uses exception-based negative testing, which in turn relies on the Lakos Rule as a design principle, are Bloomberg's BDE libraries. However, Bloomberg is by no means the only company relying on this strategy. In fact, both authors of this paper work at companies that are entirely unrelated to Bloomberg and whose codebases make extensive use of exception-based negative testing, rely on the Lakos Rule, and would be unable to effectively test their code without it. In this section, we discuss our own experience with using the Lakos Rule in practice.

3.1 Timur Doumler: *Cradle*

In 2018, I cofounded the music technology company Cradle (`https://cradle.app`) and became its CTO. I was in the enviable position of being able to start a brand new codebase from scratch, following the latest and best engineering practices and hiring a new team of developers that shared our vision.

From the start, the core guiding principle for building Cradle's software stack and engineering culture was a strong focus on code quality. One of the principles we introduced to achieve this goal was to aim for a very good unit test coverage. For whatever reason, focusing on automated testing in general and unit testing in particular tends to be less common in music production software than in other industries. We learned in practice that, by having a strong culture of unit testing and test-driven development (TDD), we were able to deliver software at a higher quality standard, with far fewer bugs and crashes reported by users.

The parts of our codebase where TDD proved to be particularly effective were the foundational, generic C++ libraries that the rest of the codebase relied upon. In particular, testing our code for contract violations (i.e., negative testing) has proven to be an important part of keeping our code quality high and reducing the number of newly introduced bugs.

As we started practicing negative testing, however, we immediately ran into the problems discussed in section 2 above. We experimented with death tests (which our chosen unit testing framework didn't offer), POSIX signals, `setjmp` and `longjmp`, and making `noexcept` conditional on whether we are in unit test mode. We found that exception-based negative testing, when combined with the Lakos Rule as a library design principle, is the most straightforward and effective method for our use case (i.e., C++ libraries for cross-platform audio software that should run — and therefore be tested on — macOS, Linux, and Windows). All alternative approaches we explored had worse tradeoffs and were ultimately nonviable for our use case.

While researching this topic, I asked C++ developers from other companies, including the maintainer of the unit testing framework we were using at the time, about negative testing. According to many of them, negative testing was "not a thing", "outside of the realm of unit testing", and so on. I found this attitude surprising, as I had proof from my own experience that negative testing can be very effective at preventing real bugs. The only explanation I can think of is that, with the Lakos Rule not being as widely used outside of the C++ Standard Library, many C++ developers have been taught to sprinkle `noexcept` all over their codebase (see also section 5), which makes negative testing very difficult, slow, and cumbersome. This abuse of the `noexcept` specifier, in turn, means that many developers never get to discover the benefits of practicing negative testing and thus are unaware of them. Consider also that many C++ developers work in smaller companies or startups that do not have the resources to develop their own unit testing frameworks (and ideally should not have to).

3.2 Ed Catmur: *Maven*

At Maven Securities (`https://www.mavensecurities.com/`), we use C++ to develop in-house software for trading on financial markets. The codebase has always been written to a high level of quality, but as the company has grown and broadened geographically, testing has become ever more crucial to maintaining a low defect rate while enabling programmers from a wide diversity of backgrounds to contribute to shared libraries in a spirit of open collaboration.

The unique requirements of the finance industry often require us to write specialised versions of Standard components (such as containers, having fine-tuned performance, latency, or memory characteristics) yet retain API compatibility (as closely as possible) with Standard and open-source libraries. This approach allows us to perform drop-in replacement of our code such that it stays readily comprehensible to coworkers, other teams, and new hires.

While negative testing is particularly prevalent in our foundational libraries, we also find it useful in higher-level components. In our line of business, warranting that bugs in market-facing code can be quickly detected and addressed during development is essential. Should defects reach production, we must also guarantee that the behaviour in the presence of defects is predictable and fail-safe and that diagnostics resulting from failure are genuinely useful to front-line support.

In particular, we approach `noxecept` in a spirit of wariness; while it has some algorithmic performance benefits in theory, the most performance-sensitive code is inlined and allocation-free and thus is highly unlikely to benefit in practice (see also section 5). On the other hand, the potential for `noexcept` to convert an exception to program termination makes widespread use of `noexcept` highly unsafe; a trading program that encounters a fault, throws an exception out to the main I/O loop, and shuts down safely is much preferred to a process that terminates immediately. Such abrupt termination potentially leaves connections in an open state and open orders on the exchange, along with exposure to financial hazard and regulatory penalties. In this context, the Lakos Rule feels entirely natural: functions having narrow contracts are either inlined, in which case `noexcept` is largely irrelevant, or they are not, in which case — even if exception-free initially — the code is unlikely to stay that way through development.

Although we use death tests where unavoidable, we find the overhead (roughly 1000-fold for fork-wait on Linux compared to throw-catch) to be a considerable impediment to achieving the code coverage and the rapid test-develop cycle to which we aspire. Additionally, the impracticality of fork-wait on Windows means that code using such tests lacks full coverage across the compilers and platforms we target.

In my experience, developers arriving at Maven and encountering our codebase for the first time are at least appreciative of the low defect rate that negative testing allows us to achieve and usually, whether from a background in the finance industry or from outside, keenly adopt the tooling that our framework provides for negative testing. To me, this anecdotal evidence indicates that the Lakos Rule is readily comprehensible, at least to developers who have seen the benefits it enables.

4 Why we need the Lakos Rule in the C++ Standard Library

Despite the usefulness of the Lakos Rule in real-world codebases, [P1656R2] argues that it should no longer be applied to the specification of the C++ Standard Library itself because existing major implementations of the C++ Standard Library do not actually use exception-based negative testing. This is an unreasonable argument, as we will demonstrate in this section.

4.1 Major C++ Standard Library implementations

Let us consider the three major implementations of the C++ Standard Library: libstdc++, libc++, and the Microsoft STL. libstdc++ and libc++ both use death tests for negative testing, while the Microsoft STL does not appear to negative test narrow contract preconditions at all.

libstdc++ chose to use death tests based on the DejaGnu framework. They have considered exception-based tests but found that they would break backward compatibility.

libc++ initially chose to use exception-based tests for ease of testing and other reasons but ran into the familiar issue that they could not apply this technique to functions declared `noexcept`. Since they could not remove `noexcept` due to backward compatibility, they introduced the conditional `noexcept` macro `_NOEXCEPT_DEBUG`, as described in section 2.1.2. They later found that `_NOEXCEPT_DEBUG` was a "horrible decision" (see [LLVMReviewD59166]) because it was observable to the user and changed the behaviour of the program. Left with no other option, they switched to fork-based death tests, which are much slower and run only on UNIX-like platforms.

This anecdote does not demonstrate that exception tests are a bad thing but rather that if they are to be used, the library should be designed for their use from the start. The corollary is that if library implementors (especially any other than the three major ones) are restricted to using death tests, as would be the result of [P1656R2], they would be able to fully test only on UNIX-like platforms (no Windows, no bare metal, no browser). Adopting [P1656R2] would do irreversible damage: if we were to reverse such a decision in the future, drawing any benefit would be difficult since users would have already come to depend on functions having narrow contracts being declared `noexcept`.

4.2 Nonmajor and non-Standard implementations

In addition to the three major implementations, a number of nonmajor implementations as well as quasi-implementations are available: libraries that do not implement the C++ Standard Library in its entirety, but a subset of it, or a superset of a subset. Libraries like Bloomberg's BSL, Electronic Arts' EASTL, NVIDIA's C++ Standard Library, and others fall into this category.

Beyond that, many more C++ libraries do not claim to be "standard" libraries but implement drop-in replacements for certain parts of the C++ Standard Library. Often, they differ in implementation to account for industry-specific requirements but follow the Standard API as closely as possible for compatibility. We have such libraries at Cradle, providing alternative implementations of containers, algorithms, allocators, and more; many companies relying on C++ have similar libraries.

Many of these libraries use exception-based testing and rely on the Lakos Rule. If the C++ Standard Library changes its design guideline in this regard, those libraries will have the choice between either having an API that is no longer following the design of the Standard or moving away from exception-based negative testing. In practice, the latter means either switching to death tests (which, as discussed, introduces a lot more complexity and overhead and, in many cases, is outright impossible) or giving up on negative testing entirely (which significantly reduces test coverage and compromises code quality).

4.3 *Throws: nothing* vs. `noexcept` as a design guideline

Note that the C++ Standard allows implementations to unilaterally tighten *Throws: nothing* to `noexcept` if they so choose — and some do so — and still be conforming. Therefore, abolishing the Lakos Rule in the C++ Standard Library specification would do all the aforementioned damage to users relying on it, while not actually benefitting anyone. If declaring functions having narrow contracts `noexcept` provides a positive tradeoff for a particular implementation of the C++ Standard Library, it can continue to do so without changing the status quo.

[P1656R2] claims that the difference between specifying *Throws: nothing* in the C++ Standard and specifying `noexcept` in a particular implementation that chooses to tighten the specification is surprising to users and somehow compromises the design of the C++ Standard. This claim is unfounded. If the difference causes confusion, clarity can and should be provided through consistency, QoI, documentation, and education. The Lakos Rule is highly motivated and straightforward to explain and understand. One of several ways to motivate it, "so we can throw exceptions to test debug-mode asserts", contains just ten words. We should not compromise the ability to test implementations on diverse platforms — a real benefit that prevents bugs in production software — for a perceived cleanliness of design.

Other ways to motivate the Lakos Rule are completely unrelated to negative testing and contract checking. A function having a wide contract that is known to never throw (given that it is implemented correctly) can be declared `noexcept`. In contrast, the behaviour of a function having a narrow contract is undefined when called out of contract; the C++ Standard does not place any restrictions on the behaviour in this case, including any restriction to not throw. We can therefore conclude that it is not logically sound to declare such a function `noexcept`: Doing so would implicitly define behaviour beyond the current domain of the function and, hence, that extended behaviour would no longer be undefined (according to the letter of the Standard). The concept of a narrow contract and that of a `noexcept` function therefore contradict each other (see [P2861R0] for a detailed discussion).

This has direct consequences for software design. It follows that once a function having a narrow contract (for example, `std::span::operator[]`, which has undefined behaviour when called out of bounds), is declared `noexcept`, it can never be backward-compatibly extended to having a wide contract (for example, be made bounds-safe in a future version by throwing when called out of bounds) due to the implicitly defined behaviour of the `noexcept` specifier.

The C++ Standard Library should be an example of sound C++ library design. Abandoning the Lakos Rule would go directly against this goal.

5 When should we use `noexcept`?

5.1 Code size and performance

The Lakos Rule stipulates that functions having a narrow contract should not be declared `noexcept`, even if they are known to never throw an exception when called in contract. Part of the resistance to this rule is a widespread practice to declare as many functions as possible `noexcept`, often for no good reason.

In some cases, `noexcept` can measurably reduce the size of the generated binary code. Such a reduction might occur when the compiler cannot otherwise reason about the function not throwing (for example, because its definition is in another translation unit). In particular, when calling a non-`noexcept` function `f` from a `noexcept` function, the compiler has to ensure that `std::terminate` gets called when an exception gets thrown (and escapes the calling function). In general, that means that instead of `f()`, the compiler generates

```
try { f(); } catch ( ...) { std::terminate(); }
```

On the other hand, when calling a `noexcept` function from another `noexcept` function, the compiler can emit just the function call (if we ignore inlining). In addition, for a `noexcept` function, the compiler does not have to generate unwind information because such a function never participates in unwinding.

Older platforms do exist, notably including 32-bit Windows, for which generating unwind information has a runtime cost (see [TR18015] section 5.4, where such platforms are said to use the "code" approach). On most platforms however, generating unwind information happens at compile time (the "table" approach). This is also known as the *zero-overhead exception model* and has become the de facto standard for essentially all modern 64-bit architectures (see [Mortoray2013]).

On such platforms, the differences in codegen between `noexcept` and non-`noexcept` typically lead to no measurable (let alone significant) difference in runtime performance (for a detailed discussion, see "Unrealizable runtime performance benefits" within the "`noexcept` Specifier" section of [EMC++S]). In fact, we are unaware of any study showing a measurable speedup in real-world code on any modern platform due to `noexcept`. Similarly, we are unaware of any study showing that exception-handling codegen has any penalty to compiler optimisations (as is sometimes claimed). [Mahaffey2017] and [Dekker2019] even found that `noexcept` can cause a net performance loss in certain cases. This loss is typically due to code motion across cache lines that can produce noise in either direction; this noise usually far outweighs any other impact of `noexcept` on performance.

More compact codegen can, of course, be a benefit in itself, even if there is no speedup whatsoever, particularly on embedded platforms where small binary size is an important concern. But on such platforms, C++ is typically compiled with exceptions disabled anyway, which removes any potential benefit from adding `noexcept` to function declarations.

Note also that in performance-critical code the affected functions in the hot path will typically be inlined. Even if exceptions are enabled, and if there are optimisations that the compiler can perform based on the function not throwing, it will be able to perform these optimisations anyway if the function is inlined, even if it is not declared `noexcept`.

5.2 The actual use case for `noexcept`

There is one genuine reason to declare a function `noexcept`: When a C++ program programmatically queries whether a function can throw, using the `noexcept` operator, and then chooses a different algorithm depending on the return value of that operator.

An example of an algorithm where such a query occurs is `std::vector::push_back`. Typically, in the presence of `noexcept`, copies will be turned into more efficient moves, which is both an observable change in behaviour and a measurable difference in performance. This is the original motivation for introducing `noexcept` in C++11 (see [N2855] and [N3050]), and its introduction is tightly linked to the introduction of move semantics. The functions being queried with the `noexcept` operator are nearly always copy, move, or swap operations. Hence, we would expect use of `noexcept` to be limited to copy and move constructors, copy and move assignment operators, and implementations of `swap`. Of these, only `swap` has a narrow contract (it requires equal allocators).

It follows that `swap` is the only bona fide exception to the Lakos Rule. We do not see a good reason to deviate from the Lakos Rule in any other cases, even in performance-sensitive code, unless a measurement can prove otherwise. This way, we can continue to enable usage of the effective exception-based negative testing strategy for the vast majority of functions having narrow contracts and require a fallback to death tests or other alternatives *only* in those vanishingly few cases (e.g., copy, move, and swap) where there is a sound engineering reason to declare the function under test `noexcept`.

Looking beyond negative testing, we should make sure that exceptions continue to be well supported and optimised by the platforms and libraries we depend on. `noexcept` has a tendency to be overused, and if exceptions keep hitting arbitrary `noexcept` barriers, they are likely to rapidly reduce in usability.

6 Can Contracts make the Lakos Rule obsolete?

SG21 is currently working on standardising a *Contracts facility* — i.e., a new language feature to be added to the C++ Standard — that allows the user to express preconditions, postconditions, and assertions in C++ code. Having a language-based Contracts facility would have many advantages over current library-based approaches such as the `ASSERT` macro that we used in section 2 above.

Attempts to standardise a Contracts facility have a long history. The design in [P0542R5], sometimes called “C++20 Contracts”, almost made it into C++20 but was removed from the working draft at the last minute because of lack of consensus on some aspects of the design. After this failure to standardise Contracts for C++20, SG21 was established and is currently aiming to get a Contracts

MVP into C++26. See [P2695R1] for the current SG21 roadmap as well as [P2521R3] and references therein for a summary of the current state of this effort.

The current Contracts MVP proposes two build modes: *No_eval*, in which the precondition is ignored, and *Eval_and_abort*, in which the precondition is checked; if the predicate evaluates to `false`, `std::terminate` is called. Note that such an MVP does not yet give us anything useful for the purposes of negative testing. Calling `std::vector::front` out of contract in *No_eval* mode is not diagnosable at run time; in *Eval_and_abort* mode, an out-of-contract call will result in `std::terminate` being called, which leaves death tests as the only method to write tests for such a call.

However, the Contracts MVP is a work in progress. SG21 is currently working on adding violation handling to the Contracts MVP. A recent proposal, [P2811R3], allows the user to install a custom violation handler at link time. Among other things, such a violation handler might be specified to throw an exception. This would give us a standard mechanism to perform exception-based negative testing.

While this would be a great outcome, note that according to all current proposals in this space (see [P2698R0], [P2811R3], and [P2834R0]), neither a violation handler nor the contract-checking predicate itself should be allowed to throw through a `noexcept` boundary. An attempt to do so would call `std::terminate` as is the case today. On platforms where death tests are nonviable (see section 2.6), the Lakos Rule will therefore still be required to conduct negative testing, even after adding a Contracts facility to the C++ Standard. We should therefore not remove the Lakos Rule as a design guideline for the C++ Standard Library.

7 Conclusion

Testing code for contract violations (negative testing) is an important part of keeping code quality high and reducing the number of introduced bugs. This approach is well proven in practice. Out of all implementation strategies for negative testing, we found that exception-based testing in combination with the Lakos Rule is the most straightforward, effective, and portable.

We have considered alternatives that do not require the Lakos Rule, such as a conditional `noexcept` macro, `setjmp` and `longjmp`, using child threads, signals, and three different flavours of death tests. All of them have unfavourable tradeoffs: they either do not scale due to an unacceptable performance overhead, are not implementable on all relevant platforms, or are outright incapable of providing the necessary functionality. In particular, the only alternatives to the Lakos Rule that seem to be somewhat viable are fork-based and clone-based death tests but *only* for UNIX-like platforms (and at reduced efficiency); for other platforms, there are none.

Some C++ Standard Library implementations choose to flout the Lakos Rule and declare nonthrowing functions having narrow contracts `noexcept`. This practice is due to a combination of having to maintain backward compatibility, not caring about non-UNIX-like platforms (which means they can use death tests instead of exception-based tests, albeit at the price of higher complexity, worse performance, and other tradeoffs), or not caring about testing for contract violations at all. For these implementations, being unable to use exception-based testing is a choice they are free to make: replacing *Throws: nothing* by `noexcept` is perfectly Standard-conforming, and they can continue to do so without changing the status quo.

Removing the Lakos Rule as a design guideline, however, would preclude the entire C++ community from using exception-based testing for Standard-conforming APIs. This regression would affect not only the major implementations of the C++ Standard Library, but also minor implementations, partial or modified implementations that are industry-specific or platform-specific, and the many non-Standard libraries that implement drop-in replacements with Standard-conforming APIs. Thus, removing the Lakos Rule would irreparably break existing testing strategies or make the affected APIs no longer Standard-conforming, while not providing any practical benefit to anyone. Bloomberg's BDE libraries are one well-known example of a codebase that would be negatively affected but are certainly not the only one: In this paper, we have shown case studies from two separate companies (unrelated to Bloomberg) that would suffer the same fate, and we are aware of others.

If we look beyond negative testing and consider the actual use case for `noexcept`, we arrive at the conclusion that specifying a function as *Throws: nothing* and declaring it `noexcept` are conceptually different and serve entirely different purposes (specifying a narrow contract on the one hand and choosing the most efficient algorithm that uses a copy, move, or swap operation on the other hand). More broadly, from a software design perspective, the definition of a narrow contract and that of a `noexcept` function are fundamentally incompatible (the former specifies that for *some* input, the behaviour is undefined, while the latter specifies that for *all* input, the function is defined to not throw). This causes real issues with library design: declaring a function with a narrow contract `noexcept` makes it impossible to widen the contract later without breaking backward-compatibility. Removing the Lakos Rule would mean abandoning the idea that the C++ Standard Library should follow sound, consistent design principles.

We have also considered the ongoing work toward standardising a C++ Contracts facility. We conclude that Standard Contracts could become a powerful new tool for testing code but does not make the Lakos Rule any less necessary because a language-based Contracts facility will not change the fact that an exception cannot be thrown through a `noexcept` boundary.

The Lakos Rule is a long-standing design principle of the C++ Standard Library and is highly motivated and straightforward to explain and understand. Changing such an established principle requires reaching a high bar of justification. For all the reasons discussed in this paper, this bar for removing the Lakos Rule is clearly unmet. We therefore urge the C++ Standards Committee to maintain the status quo, that is, to retain the Lakos Rule.

Acknowledgements

We would like to thank Lori Hughes, John Lakos, and Mungo Gill for thoroughly reviewing this paper and providing useful feedback; to Ville Voutilainen for describing his idea of negative testing with stackful coroutines; and to Jonathan Wakely for providing additional useful comments.

References

[Dekker2019] Niels Dekker. `noexcept` considered harmful. https://www.youtube.com/watch?v=dVRLp-Rwg0k, CppOnSea 2019.

[EMC++S] John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith. Embracing Modern C++ Safely. Addison-Wesley, 2021.

[GTestDocDeathTests] Advanced GoogleTest Topics — Death Tests. https://github.com/google/googletest/blob/main/docs/advanced.md#death-tests, 2023-03-24.

[LLVMReviewD59166] Remove exception throwing debug mode handler support. https://reviews.llvm.org/D59166, 2019-03-08.

[MSVCDocLongjmp] Microsoft Visual Studio 2022 Documentation — `longjmp`. https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/longjmp?view=msvc-170, 2022-02-12.

[Mahaffey2017] Terry Mahaffey. Please, Please Help the Compiler. https://github.com/TriangleCppDevelopersGroup/TerryMahaffeyCppTalk, 2017-12-09.

[Mortoray2013] Edaqa Mortoray. The true cost of zero cost exceptions. https://mortoray.com/2013/09/12/the-true-cost-of-zero-cost-exceptions/, 2013-09-12.

[N2855] Douglas Gregor and David Abrahams. Rvalue References and Exception Safety. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2855.html, 2009-03-23.

[N3050] David Abrahams, Rani Sharoni, and Doug Gregor. Allowing Move Constructors to Throw (Rev. 1). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3050.html, 2010-03-12.

[N3248] Alisdair Meredith and John Lakos. `noexcept` Prevents Library Validation. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3248.pdf, 2011-02-28.

[N3279] Alisdair Meredith and John Lakos. Conservative use of `noexcept` in the Library. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3279.pdf, 2011-03-25.

[O'Dwyer2018] Arthur O'Dwyer. The Lakos Rule. https://quuxplusone.github.io/blog/2018/04/25/the-lakos-rule/, 2018-04-25.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html, 2018-06-08.

[P0884R0] Nicolai Josuttis. Extending the noexcept Policy, Rev0. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0884r0.pdf, 2018-02-10.

[P1656R2] Agustín Bergé. "Throws: Nothing" should be `noexcept`. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1656r2.html, 2020-02-11.

[P1705R1] Shafik Yaghmour. Enumerating Core Undefined Behavior. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html, 2019-09-28.

[P2148R0] CJ Johnson and Bryce Adelstein Lelbach. Library Evolution Design Guidelines. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2148r0.pdf, 2020-09-23.

[P2514R0] René Ferdinand Rivera Morell and Isabella Muerte. `std::breakpoint`. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2514r0.html, 2021-12-30.

[P2521R3] Gašper Ažman, Joshua Berne, Bronek Kozicki, Andrzej Krzemieński, Ryan McDougall, and Caleb Sunstrum. Contract support – Record of SG21 consensus. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2521r3.html, 2023-02-10.

[P2695R1] Timur Doumler and John Spicer. A proposed plan for Contracts in C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2695r1.pdf, 2023-02-09.

[P2698R0] Bjarne Stroustrup. Unconditional termination is a serious problem. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2698r0.pdf, 2022-11-18.

[P2811R3] Joshua Berne. Contract-violation handlers. https://isocpp.org/files/papers/P2811R3.pdf, 2023-05-04.

[P2834R0] Joshua Berne and John Lakos. Semantic Stability Across Contract-Checking Build Modes. https://isocpp.org/files/papers/P2834R0.pdf, 2023-05-08.

[P2861R0] John Lakos. Narrow Contracts and `noexcept` Are Inherently Incompatible: The Lakos Rule. https://wg21.link/p2861r0, 2023-05-15.

[TR18015] Technical Report on C++ Performance. https://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf, 2006-02-15.