`noexcept` policy for SD-9 (The Lakos Rule)

Timur Doumler (papers@timur.audio)
John Lakos (jlakos@bloomberg.net)

Document #: P3155R1
Date: 2026-06-05
Project: Programming Language C++
Audience: LEWG

Abstract

This paper is a LEWG policy proposal as per [SD-9]. It proposes to reconfirm the Lakos Rule (a function having a narrow contract should not be declared `noexcept`), a long-standing design principle of the C++ Standard Library, as an official LEWG design policy. We briefly summarise the rationale why the Lakos Rule is still an important and valuable design principle today – and even more so now that contract assertions are in C++26. We present a quantitative study showing that the Lakos Rule is overwhelmingly followed in the existing C++ Standard Library specification, with exceptions to the rule being principled and clearly localised.

Contents

Revision history

Revision 1 (2026-06-04)

— Added prior art section

— Expanded rationale

— Rebased paper on C++26

Revision 0 (2024-04-16)

— Original version

0 Motivation and context

A long-standing design principle in the C++ Standard Library is that a function having a narrow contract – that is, a function specified to have *preconditions* – should not be declared `noexcept`, even if it is known to never throw an exception when called in-contract (i.e., without violating the preconditions). When we are convinced that a function having a narrow contract cannot throw when called in-contract, it should instead be specified as *Throws: nothing*. This design principle is also known as the *Lakos Rule*.

The Lakos Rule was first proposed in [N3248] and adopted via [N3279] in WG21 plenary ([N3279]). An updated version of the rule was later codified into LEWG policy in [P0884R0]; see [O'Dwyer2018] for a more detailed summary.

[P1656R2] proposed abandoning the Lakos Rule as a design principle for the C++ Standard Library. 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. Additionally, [P2148R0] proposed adopting a set of design policies for the evolution of the C++ Standard Library that include moving away from the Lakos Rule.

While no policy to abandon the Lakos Rule has been formally adopted, LEWG nevertheless moved away from adhering to the Lakos Rule. LEWG leadership assumed the position that no formally adopted policy exists for how preconditions and `noexcept` should interact ([P2920R0]); instead, we should consider each new proposal that contains narrow-contract functions on a case-by-case basis. This led to newer library facilities (e.g., `std::hive`) diverging from the Lakos Rule.

We believe that the Lakos Rule should remain a design principle for the C++ Standard Library. There are important engineering reasons why the Lakos Rule is still valid and useful today; abandoning it would inflict avoidable damage to the C++ language . To argue this case, we published two papers in the May 2023 mailing, [P2831R0] and [P2861R0], and had planned to present them at the June 2023 meeting in Varna.

However, neither paper was scheduled for discussion in LEWG at the Varna meeting, nor at the following meeting in Kona in November 2023. Instead of considering our papers, LEWG has decided to first agree on a general process for adopting design policies for the C++ Standard Library. This has been done in [SD-9], with the actual policy-adoption process explained in [P2267R1].

According to this process, LEWG design policies should be adopted via *policy papers*. Such policy papers must satisfy a number of requirements in order to be considered by LEWG. [P2831R0] and [P2861R0] have not been written with those requirements in mind, because they have been published before [SD-9] was adopted, and were thus rendered ineligible for discussion in LEWG. The purpose of the present paper is to satisfy the requirements set out in [P2267R1] and to serve as an "envelope paper" for [P2831R0] and [P2861R0] to make their content admissible for a `noexcept` policy discussion in LEWG, whenever this discussion is eventually scheduled.

Another paper relevant to this discussion is [P3005R0]. Rather than proposing a concrete `noexcept` policy, it proposes a *decision-making process* for how to agree on such a policy. The process outlined in [P3005R0] involves systematic review of the entire design space: the paper considers seven possible candidate `noexcept` policies, six design questions that would need to be answered to choose one of those policies, and lists 20 design principles that can help us find the correct answers to those questions by following the *principled-design* methodology proposed in [P3004R0]. We believe that the process outlined in [P3005R0] is an appropriate way to determine the best policy for the C++ Standard Library and establish consensus on that policy in LEWG. The present paper serves to aid this effort.

1 Rationale

The rationale for retaining the Lakos Rule as a design policy has been thoroughly explored in our previous papers: [P2831R0], [P2861R0], and [P3005R0]. We therefore refer the reader to those papers and references therein for details. Here, we provide only a brief summary of the most important engineering reasons to retain the Lakos Rule:

— **Contract widening.** Once a function with a narrow contract is specified as `noexcept`, its contract can never be widened to throw on what was previously a contract violation. For example, `std::vector::operator[]` could never be made backwards-compatibly bounds-checked with the behaviour of `std::vector::at` if it was `noexcept`. This is because `noexcept` is part of the type, part of the ABI, and directly observable via `operator noexcept`. Further, `noexcept` breaks Liskov substitutability — a bounds-checked subclass can no longer substitute for a narrow-contract base. On the other hand, *Throws: Nothing* preserves the evolutionary freedom to widen the contract in a backwards-compatible and Liskov-substitutable way.

— **Logical inconsistency.** Specifying a function as `noexcept` imposes a nothrow-guarantee over the entire syntactically valid input domain, contradicting the very definition of a narrow contract (the input domain is restricted, behaviour is undefined outside of it). `noexcept` thus effectively widens the contract in a specific way; the two concepts are inherently incompatible. Using `noexcept` on a narrow-contract function also contradicts the definition of undefined behaviour itself as behaviour on which the Standard imposes no restrictions. By imposing practical restrictions on UB, `noexcept` limits implementation freedom of how to mitigate it.

— **Asymmetry of composition.** A C++ program adhering to the Lakos Rule can be written atop a library that does, but not atop one that does not. The C++ Standard Library is foundational, therefore it should follow the rule. Note that implementations remain free to strengthen *Throws: nothing* to `noexcept` unilaterally. Therefore, retaining the Lakos Rule costs nothing – on the other hand, abandoning it removes options irreversibly.

— **Programmatic recovery.** There are many applications (e.g., in gaming, desktop publishing, host-plugin systems) for which throwing an exception on precondition failure and recovering programmatically is vastly preferable to terminating the program. However, a `noexcept` barrier turns any such throw into unconditional `std::terminate`. If the Lakos Rule is abandoned, such applications are permanently prevented from using any conforming Standard Library implementation.

— **Negative testing.** The cleanest, fastest, and only portable and scalable way to test that a precondition check actually fires is to throw from the assertion handler and catch it. However, this only works if functions with preconditions are not `noexcept`. All known alternatives (death tests via fork, clone, or spawn, `setjmp/longjmp`, child threads, signals, conditional-`noexcept` macros) are variously platform-specific and non-portable, orders of magnitude slower, UB-prone, or change the code under test (which is why libc++'s `_NOEXCEPT_DEBUG` experiment was abandoned as a "horrible decision").

— **Foundational for Contracts.** The Lakos Rule is foundational for C++26 contract assertions, which were designed with the assumption that the Lakos Rule is an established design principle. C++26 allows a contract-violation handler to throw out of the function whose contract was violated, providing a standard way to implement programmatic recovery and negative testing. However, throwing contract-violation handlers do not work in the presence of `noexcept` barriers. C++26 contract assertions and `noexcept` are thus mutually exclusive if one wishes to use throwing contract-violation handlers. For a consistent language design, we need to either reinstate the Lakos Rule, or remove throwing contract-violation handlers. However,

doing the latter has been rejected multiple times in EWG, whose position is clearly that throwing contract-violation handlers belong in the language.

— **Misuse of `noexcept`.** There is no study showing measurable *runtime* performance benefits through the use of `noexcept` on any modern platform. There is meeasurable impact on *code size*, but the impact is minor, and for applications where code size really matters exceptions are typically disabled anyway. Merely documenting that a function cannot throw when called in-contract can be done through other, less risky and invasive means. The genuine use case for `noexcept` – and the only one for which it was originally designed – is to enable an algorithm such as `std::vector::push_back` to choose a different, algorithmically more efficient code path for types that are nothrow-movable/copyable/swappable. When `noexcept` usage is limited to this use case, friction with the Lakos Rule (which was developed in concert with `noexcept` for C+11) disappears entirely.

2 Prior art in the C++ Standard

The C++ Standard Library has been largely following the Lakos Rule as a design principle since `noexcept` was introduced in C++11 ([N2855], [N3248], [N3279]), but exceptions do exist. We have conducted a detailed survey of narrow-contract functions across the entire current C++ working draft [N5046] to both quantify and qualitatively describe these exceptions. We summarise the results of our survey in this section.

Counting both hardened and non-hardened preconditions, ignoring deprecated functions in Annex D, and properly accounting for member functions of concrete container types that are described in generic clauses such as [sequence.reqmts], we counted 1284 narrow-contract functions in the C++ Standard library (out of 7165 total, i.e. 17.9%). Out of these 1284 narrow-contract functions:

— 1053 functions (82.0%) are not `noexcept`, i.e. they follow the Lakos Rule unconditionally.

— 204 functions (15.9%) are `noexcept`, i.e. they violate the Lakos Rule.

— 27 functions (2.1%) are *conditionally* `noexcept`.

A pie chart visualising this distribution is shown in Figure 1.

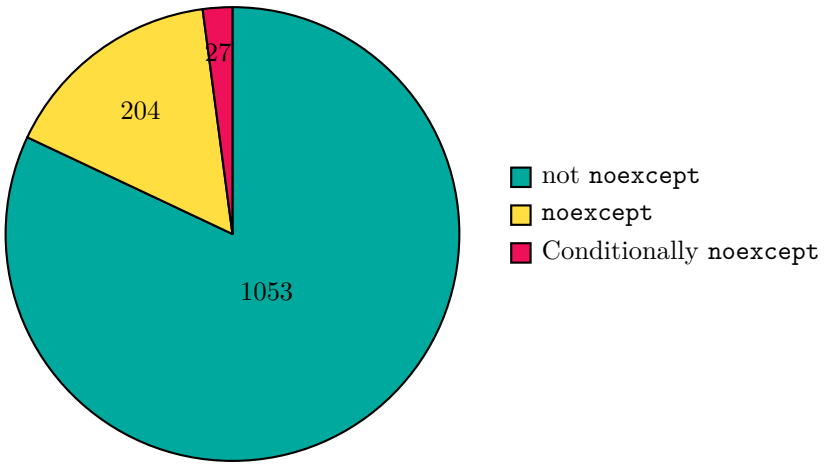

Figure 1: `noexcept`-ness of narrow-contract Standard Library functions across the current C++ working draft

Violations of the Lakos Rule are not random, but principled (at least, before C++26). Almost all narrow-contract functions that are conditionally `noexcept` are clearly localised in the following clusters, ordered from most to least common:

— Dereferencing observers and accessors in smart pointers, `std::optional`, `std::expected`, `std::span`, and `std::mdspan`;

— Low-level, concurrency-related facilities such as `std::atomic`, `std::atomic_ref`, hazard pointers, RCU;

— Deallocation functions such as `operator delete(void*)`, which are forbidden from throwing by core language rules;

— Functions that cannot throw by the very nature of their operation, even if called out-of-contract, such as `std::launder` and `std::start_lifetime_as`;

— Lakos Rule violations in facilities newly introduced in C++26, such as `std::hive` and `std::saturating_div`.

The first and fourth clusters are deliberate, well-motivated divergences from the `noexcept` policy established via [P0884R0]; the second and third are in line with the policy; the last one indicates recent divergence in the absence of a policy.

Conditionally-`noexcept` functions with narrow contracts are few. They are localised in the following two clusters:

— Swap and move operations: `std::swap`, member `swap`, `iter_swap`, `iter_move`, move-assign operators;

— Forwarding `operator()(Args...)` in function wrappers `std::move_only_function` and `std::copyable_function`.

In all of the above cases, the throwing behaviour tracks that of a template parameter, in line with the policy in [P0884R0]. This is a narrow, well-defined idiom that has been explicitly designed with the Lakos Rule in mind, carving out precise exceptions to the rule where algorithmically necessary.

Apart from the above clusters and a small handful of isolated cases, the C++ Standard Library consistently adheres to the Lakos Rule. Many popular and widely used C++ Standard Library facilities adhere to the Lakos Rule *without any exceptions*: all containers introduced before C++26 (`std::vector`, `std::array`, associative containers), all algorithms introduced before C++26, as well as `std::variant`, `std::pair`, `std::tuple`, `std::mutex`, and `std::chrono`. Other popular facilities such as `std::thread` are entirely wide-contract so the question does not arise.

The use of the "*Throws:* Nothing" specification idiom is widespread, but not rigorous – sometimes the "*Throws:*" clause is missing even if a non-`noexcept` function will not throw in-contract, such as for `operator[](size_t)` for array `std::unique_ptr`.

Our analysis demonstrates that the Lakos Rule is the norm in the C++ Standard Library today. Therefore, reinstating the Lakos Rule as official LEWG policy is the only choice consistent with the status quo in the C++ Standard Library specification.

3 Status quo in the wider C++ community

Community surveys such as the JetBrains Developer Ecosystem survey, the Meeting C++ Community Survey, and the Standard C++ Foundation's Annual C++ Developer Survey consistently show that about half of C++ developers work on codebases where exceptions are either partially or completely banned from use; for the latter in particular, the `noexcept` policy is of little relevance.

Codebases that *do* use exceptions fall into different camps. Many C++ libraries generously sprinkle `noexcept` all over their code, while codebases relying on techniques like exception-based recovery and negative testing adhere to the Lakos Rule (see [P2831R0] for case studies of several companies with such codebases, across different industries).

Quantifying the relative size of these camps is difficult and we did not attempt to do so for this paper. However, we expect the use of these techniques to become significantly more widespread with the adoption of C++26 contract assertions, which permit attaching a throwing contract-violation handler as a *standard*, portable mechanism for exception-based recovery from assertion failures.

The three major implementations of the C++ Standard Library (GCC, Clang, and Microsoft) do not use the Lakos Rule and generally tighten *Throws: nothing* to `noexcept`, making them incompatible with these techniques; other implementations of the Standard Library, or parts of it, such as Bloomberg's BDE, follow the Lakos Rule and make extensive use of exception-based recovery and negative testing.

4 Proposed policy

We propose to retain the `noexcept` policy that we have already had for a long time and that the vast majority of the C++ Standard Library follows today: the one described in [P0884R0], refining the original formulation in [N3279]. This policy consists of the following rules, which includes the Lakos Rule:

a) No library destructor should throw. They shall use the implicitly supplied (nonthrowing) exception specification.

b) If a library function has a wide contract (i.e., does not specify undefined behavior due to a precondition), it should be marked as unconditionally `noexcept` if it cannot throw; if it has a narrow contract, it should be specified as *Throws: Nothing* if it cannot throw when called in-contract.

c) If a library swap function, move-constructor, or move-assignment operator is conditionally-wide (i.e. can be proven to not throw by applying the `noexcept` operator) then it should be marked as conditionally `noexcept`.

d) If a library type has wrapping semantics to transparently provide the same behavior as the underlying type, then default constructor, copy constructor, and copy-assigment operator should be marked as conditionally `noexcept` matching the underlying exception specification.

e) No other function should use a conditional `noexcept` specification.

f) Library functions designed for compatibility with C code (such as the atomics facility) may be marked as unconditionally `noexcept`.

Note that this set of rules corresponds to Policy C in [P3005R0], which considers seven possible `noexcept` policies B-H in addition to the null policy A ("no policy").

5 Rationale for adopting a policy

The question of which `noexcept` policy the C++ Standard Library should follow has been a hotly debated topic in recent years, taking up a considerable amount of committee time. Equally, per-proposal per-function re-litigation of `noexcept`-ness – required in the absence of any agreed-upon policy, which seems to be the status quo – is also an inefficient use of committee time.

While no official policy exists, the C++ Standard Library continues to drift apart on this question, with the bulk of it adhering to the Lakos Rule as originally intended but newer (C++26) facilities departing from it. This creates inconsistencies and growing technical debt that will need to be retroactively fixed via Defect Reports once we have an agreed-upon policy, which will consume even more committee time.

For all of the above reasons, we should put this debate to rest and adopt a `noexcept` policy as soon as possible.

6 Proposed wording

Append to "List of Standard Library Policies" section of SD-9 the items a) – f) enumerated in Section 4 above.

Disclaimer

The quantitative survey of Lakos Rule adherence across the entire C++ working draft has been performed with the help of an LLM. The exact prompts used for this survey and their full output are available from the authors upon request.

Acknowledgements

We wish to express our gratitude to the late Ed Catmur, co-author of [P2831R0], for his valuable contributions to this effort until his tragic death during a fell run on 31 December 2023.

References

[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.

[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.

[N5046] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://wg21.link/n5048, 2026-05-12.

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

[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.

[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.

[P2267R1] Inbal Levi, Ben Craig, and Fabio Fracassi. Library Evolution Policies — The rationale and process of setting a policy for the Standard Library. https://wg21.link/p2267r1, 2023-11-23.

[P2831R0] Timur Doumler and Ed Catmur. Functions having a narrow contract should not be `noexcept`. https://wg21.link/p2831r0, 2023-05-15.

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

[P2920R0] Bryce Adelstein Lelbach, Nevin Liber, Robert Leahy, Ben Craig, Fabio Fracassi, and Guy Davidson. Library Evolution Leadership's understanding of the `noexcept` policy history. https://wg21.link/p2920r0, 2023-06-16.

[P3004R0] John Lakos, Harold Bott, Mungo Gill, Lori Hughes, Alisdair Meredith, Bill Chapman, Mike Giroux, and Oleg Subbotin. Principled Design for WG21. https://wg21.link/p3004r0, 2024-02-13.

[P3005R0] John Lakos, Joshua Berne, Harold Bott, Mungo Gill, Alisdair Meredith, Bill Chapman, Mike Giroux, and Oleg Subbotin. Memorializing Principled-Design Policies for WG21. https://wg21.link/p3005r0, 2024-02-15.

[SD-9] Inbal Levi. Library Evolution Policies. https://wg21.link/sd9, 2024-05-14.