P2946R0: A Flexible Solution to the Problems of `noexcept`

Pablo Halpern <phalpern@halpernwightsoftware.com>

2023-07-19 18:21 EDT

Target audience: EWGI

1 Abstract

The `noexcept` function specifier and companion `noexcept` operator were invented to allow containers such as `vector` to provide the strong exception-safety guarantee when performing operations that require relocating existing elements. The `noexcept` specifier has since been utilized regularly to improve code generation and sometimes as a form of documentation. The problems with these off-label uses have been known for a long time; the *Lakos Rule*, which predated the release of C++11, was intended to prevent such use (or misuse) within the C++ Standard Library. This paper proposes an attribute, `[[throws_nothing]]`, as an alternative to `noexcept` for annotating nonthrowing functions. Being invisble to the `noexcept` operator and having implementation-defined semantics, `[[throws_nothing]]` is a less powerful annotation than `noexcept` that avoids the issues the Lakos Rule was created to address. Thus, `[[throws_nothing]]` eliminates the temptation to abuse `noexcept` by providing a tool better suited for improving code generation and documenting programmer intent.

2 Brief Sketch of the Proposed Feature

This minimal description of the `[[throws_nothing]]` feature is sufficient for understanding the "Motivation" section below. A full description of this proposal is found in the "Proposed Feature" section.

A new fuction attribute, `[[throws_nothing]]`, is proposed for annotating a function that does not throw when called within a correct program. The attribute is not detectable using the `noexcept` operator:

```
[[throws_nothing]] void f(int);
static_assert(noexcept(f(0)) == false);
```

Whether the program will terminate if an exception attempts to escape from `f`, above, is implementation defined (and thus potentially user configurable).

3 Motivation

3.1 Purpose of `noexcept`

The `noexcept` specifier was introduced at the end of the C++11 cycle for one purpose: to enable the safe use of move constructors in `vector`-like containers that offer the *strong exception-safety guarantee* for certain operations, such as inserting elements at the end. The problem was first described, with `noexcept` as the proposed solution, in N2855. This proposal was later refined, eventually resulting in the final wording of N3050.

Below is one possible implementation of a vector-reallocation function, which must leave the original vector unchanged if an exception is thrown while trying to move elements from the old buffer to the new one.

This implementation uses `if constexpr` instead of the `std::move_if_noexcept` function to make the two different code paths easier to distinguish.

```
template <class T, class A>
void vector<T,A>::reallocate(size_type new_capacity)
{
  using alloc_traits = allocator_traits<A>;

  pointer new_data = alloc_traits::allocate(new_capacity);

  size_type i = 0;
  if constexpr (noexcept(T(std::move(m_data[i])))) {
    for (i = 0; i < size(); ++i)
      alloc_traits::construct(&new_data[i], std::move(m_data[i])); // efficient
  }
  else try {
    for (i = 0; i < size(); ++i)
      alloc_traits::construct(&new_data[i], m_data[i]);  // copy (less efficient)
  }
  catch (...) {
    while (i)
      alloc_traits::destroy(new_data[--i]);
    alloc_traits::deallocate(new_data, new_capacity);
    throw;
  }

  // got here only if no exception was thrown
  for (i = 0; i < size(); ++i)
    alloc_traits::destroy(m_data[i]);
  alloc_traits::deallocate(m_data, m_capacity);

  m_data     = new_data;
  m_capacity = new_capacity;
}
```

The use of `T`'s move constructor can often yield algorithmic performance advantages over using its copy constructor, sometimes reducing the cost from *O(N)* or worse per copy to *O(1)*. Such a move constructor, however, typically modifies the original object; if the move constructor might throw, `vector` must degenerate to using the copy constructor, and thus give up the performance gain to ensure that it can leave the original object in its initial state. Because `vector<T, A>::reallocate` is a *generic* function, using `std::move` and retaining the strong guarantee in the code above would be impossible if we did not have the `noexcept` operator.

3.2 The Lakos Rule

Since the `noexcept` annotation was added late in the C++11 cycle and was thus brand new and not fully understood, applying it appropriately in the Standard Library was a challenge. John Lakos and Alisdair Meredith proposed what has become known as the *Lakos Rule* (described in N3279 and extended in P0884). Summarized below, the Lakos Rule provided a conservative framework for deciding whether a specific function may safely be declared `noexcept`.

- If a function has no preconditions (a *wide contract*) and is guaranteed not to throw (via an explicit "*Throws*: nothing" clause), it may be declared `noexcept`.
- If a function has preconditions (a *narrow contract*) or if it might throw when called correctly (*in contract*), it must *not* be declared `noexcept`.

The example below shows a subset of the `std::vector` interface. Note that only `size()`, which promises not to throw and has no preconditions, is declared `noexcept`, whereas the others each fail one or both of the Lakos Rule tests and are thus not `noexcept`.

```
template <class T, class A>
class vector {
   // ...
   constexpr size_type size() const noexcept; // wide contract,   doesn't throw
   constexpr reference at(size_type);         // wide contract,   might throw
   constexpr reference operator[](size_type); // narrow contract, doesn't throw
   constexpr reference front();               // narrow contract, doesn't throw
};
```

3.3 Resistance to the Lakos Rule

Although the Lakos Rule is effective and has strong theoretical and practical underpinnings (e.g., enabling certain backward-compatible extensions and conforming wider interfaces, see P2861), two reasons have emerged for violating it.

1. Under many (but by no means all) circumstances, calling a `noexcept` function generates less code. Thus, programmers — both within and outside WG21 — want to use `noexcept` to improve code generation, yet the author has seen no compelling evidence that `noexcept` produces measurably *faster* code on any modern platforms.

2. Within WG21, concern has been voiced that the distinction between "*Throws*: nothing" and `noexcept` is unclear (see P1656).

As tempting as it might be, violating the Lakos rule is ill advised unless a compelling case can be made that querying the function with the `noexcept` operator is necessary for optimizing or ensuring correctness of a generic algorithm at compile time. As described in P2861, if `noexcept` is added to a function in one version of the Standard, it cannot be removed in a future version without potentially breaking code. Specifically, widening the contract of a function to add new functionality is safe, provided that every program written for the old version has the same observable behavior when compiled with the new version, but if the old version is annotated with `noexcept`, the new version cannot be widened to accept new values that would result in an exception being thrown.

Moreover immediate forced termination is not option in some environments. In such an environment, a defensive programming library (or language contract facility) might want to throw an exception on a precondition violation — even within a function that would not otherwise throw — so that the program can shut down gracefully or even soldier on. This ability to continue after a logic error has been detected is especially useful when testing the precondition checks themselves. The `noexcept` specifier interferes with such a throwing defensive-programming facility (see P2831R0).

3.4 Serving the C++ Multiverse

The goal of this proposal is to address the constituencies within the different C++ universes (the C++ multiverse) that have been ill served by `noexcept` alone, such as embedded-software developers who want smaller code and those who need to avoid immediate termination of their programs (e.g., upon detecting contract violations). What is needed is a way to provide the desired code-generation and documentation benefits of `noexcept` without violating either the spirit or the letter of the Lakos Rule.

3.4.1 The embedded-software development universe

The reduction in generated-code size that usually results from annotating a called function with `noexcept` or `[[throws_nothing]]` has the most impact in memory-constrained environments. WG21 members often assert that embedded-software developers turn off exceptions in their builds because the code-size cost of leaving them enabled is too large. Unfortunately, this assumption has led to a self-fulfilling prophecy: WG21 does nothing to make exceptions friendlier for embedded programmers, so embedded programmers, in turn, eschew exceptions.

Since even an embedded microprocessor may have several megabytes of RAM available to it, completely turning off exceptions is not always necessary. Given an appropriate C++ implementation, judicious use of the `[[throws_nothing]]` attribute can help an executable stay within its memory budget while following the best design practices, including the Lakos Rule.

3.4.2 The graceful-termination universe

In high-data-integrity environments, it is often unacceptable to terminate suddenly when encountering an error. To avoid data corruption, resources must be released, transactions rolled back, and user data saved, before aborting. If any of these actions would normally occur in a destructor of an RAII object, then such a graceful shutdown could not be accomplished readily in a terminate handler.

Imagine a defensive-programming library comprising an `assert`-like macro and a custom function to handle assertion failures:

```
void assert_failure_handler(const char* file, unsigned line, const char* func,
                            const char* expression)
{
  std::cerr << file << ':' << line << ": in function " << func
            << ": assertion failure " << expression << std::endl;

  throw assert_failure_exception(file, line, func, expression);
}

#ifdef CHECKED_MODE
# define ASSERT(cond) cond ? (void) 0 : \
    assert_failure_handler(__FILE__, __LINE__, __FUNCTION__, #cond)
#else
# define ASSERT(cond) (void) 0
#endif
```

Now, imagine an integer absolute-value function, `intAbs`, having the precondition that the input is not `INT_MIN`, because the absolute value of `INT_MIN` is not representable in an `int`. When called in contract, `intAbs` does not throw an exception, so it is declared with `[[throws_nothing]]`. Within the function body, `intAbs` checks its precondition using the above `ASSERT`:

```
// Return absolute value of x.  Precondition: x is not `INT_MIN`.
[[throws_nothing]] int intAbs(int x)
{
  ASSERT(x != INT_MIN);  // precondition check
  return x < 0 ? -x : x;
}
```

Code using this function might have a subtle bug that is detected only during beta testing (when real user data is at stake):

```
    int thing = read_thing();
    // Oops! Forgot to sanitize our input!
    thing = intAbs(thing);  // Precondition check might fail.
```

To trigger the precondition check without suddenly terminating the program, `[[throws_nothing]]` must allow the assert-failure exception to escape. The organization would thus choose a C++ implementation that ignores `[[throws_nothing]]`, thus allowing exceptions to propagate. By disabling the enforcement of `[[throws_nothing]]`, only the behavior of erroneous code is changed; the essential behavior is unaffected, although larger code size might be observed.

3.4.3 The must-not-terminate universe

Some programs must not terminate at all, ever. For example, a game engine might continue running its main event loop after an error is detected, even if continuing would result in a momentary glitch on the screen. The universe of such programs has similar requirements to the graceful-termination universe; unexpected exceptions thrown from presumably nonthrowing functions should not terminate the program but release resources in an orderly way, then continue.

Test drivers are an important subset of the must-not-terminate universe. A precondition check, like any other aspect of a function, should be tested — i.e., by providing inputs at the boundaries of the precondition — including deliberately violating it. When every precondition violation causes program termination, writing a portable and efficient test driver is not possible, as described in P2831R0.

Given the defensive-programming library and `intAbs` function from the "The graceful-termination universe" section above, a white-box unit test for `intAbs` could test that the `ASSERT` correctly encodes the documented precondition. Using a throwing failure handler (as shown in `assert_failure_handler`), the test engineer would write a negative test that deliberately violates the precondition:

```
bool testPreconditionViolation()
{
  try {
    intAbs(INT_MIN);
    return false; // failed to catch the precondition violation
  }
  catch (const assert_failure_exception&) {
    return true;  // successfully caught the precondition violation
  }
}
```

The test engineer would, as in the previous section, choose a C++ implementation that ignores `[[throws_nothing]]`, thus allowing the precondition check to detect the deliberate error without terminating the test program. Because `[[throws_nothing]]`, unlike `noexcept`, cannot be used to change the program logic within the caller, test engineers can have reasonable confidence that they are fully testing the function, even if the final program is eventually deployed using an implementation that terminates on a `[[throws_nothing]]` violation.

3.4.4 The library-specification universe

A number of features in C++ that are intended to reduce errors or improve code generation have the side effect of making code more self-documenting. For example, `const` indicates — to both the compiler and human reader — that a variable's value will not change, and the `assert` macro documents an invariant of an algorithm in a way that is enforceable at run time. Similarly, `[[throws_nothing]]` indicates, at a glance, that a function will not throw when called in contract; both the implementation and the human reader benefit.

Within the C++ Standard Library, functions having "*Throws*: nothing" as part of their description could be annotated with `[[throws_nothing]]`. Whether such a practice would add clarity is a matter for LWG to decide.

4 Proposed Feature

A Standard attribute, tentatively named `[[throws_nothing]]` and appertaining to function declarations, is proposed to indicate that a function is specified not to throw when all its preconditions are met (i.e., it is called *in contract*):

```
[[throws_nothing]] void f(int);
```

The presence of the `[[throws_nothing]]` attribute cannot be queried by the program itself at compile time; the result of the `noexcept` operator and the function type are unchanged:

```
[[throws_nothing]] void g1(int);
static_assert(noexcept(g1(0)) == false);

[[throws_nothing]] void g2(int) noexcept;
static_assert(noexcept(g2(0)) == true);

[[throws_nothing]] void g3(int) noexcept(false);
static_assert(noexcept(g3(0)) == false);

void g4(int);
static_assert(std::is_same_v<decltype(g1), decltype(g4)>);
```

Intentionally making `[[throws_nothing]]` invisible to the `noexcept` operator prevents using `[[throws_nothing]]` to select an algorithm at compile time; the attribute does not change the *essential behavior*[1] of a correct program and can be removed from a subsequent version of a function, provided the behavior of the function does not change for any previously valid inputs.

If a `[[throws_nothing]]` function attempts to exit via an exception, then whether `std::terminate` is called or the annotation is ignored (and the exception propagates normally) is implementation defined. The recommended best practice is to make both semantics available to the user. If, however, the function is also annotated with `noexcept` or `noexcept(true)`, `std::terminate` is always called, regardless of the implementation's semantic for `[[throws_nothing]]`.

By making the behavior of an incorrect program — one that attempts to throw from a `[[throws_nothing]]` function — implementation defined, rather than always terminating, the behavior can vary to serve the multiple constituencies of the C++ multiverse. On an implementation that calls `std::terminate`, a call to a function annotated with `[[throws_nothing]]` is likely to result in smaller generated code compared to one with no annotation at all. Conversely, an implementation that ignores the attribute allows for graceful shutdown, log-and-continue semantics, and effective testing of contract checks in functions that would not otherwise throw.

As with `noexcept` currently, implementations of the Standard Library would be permitted to use `[[throws_nothing]]` for any nonthrowing function, even though the Standard itself would never mandate its use. In fact, for discretionary use by implementations, `[[throws_nothing]]` is much better than the `noexcept` specifier because `[[throws_nothing]]` cannot inadvertently change the meaning of a correct program and is responsive to the settings used to build the program.

4.1 Feature comparison

For functions that promise not to throw, the table below compares `[[throws_nothing]]` to `noexcept` and to using no annotation at all (*unmarked*). *If terminate* means *yes* for implementations that terminate on unexpected exceptions and *no* otherwise. *If ignore* means *yes* for implementations that ignore the annotation and *no* otherwise.

The purpose of the table is not to show that one annotation is better than the other, but that, despite some overlap, they serve different purposes and therefore support different use cases, none of which violate the Lakos Rule.

[1]Essential behavior comprises the promised behavior of a function when called in contract. The return value, guaranteed side effects, and complexity guaranties are part of essential behavior. The layout of objects, number of instructions executed, and logging are rarely part of a function's essential behavior. The effects of calling the function out of contract are *never* part of essential behavior.

	unmarked	noexcept	[[throws_nothing]]
Makes function self-documenting	no	yes	yes
Provides codegen hint to compiler	no	yes	if terminate
Terminates on unexpected exception	no	yes	if terminate
Suitable for wide contracts	yes	yes	yes
Suitable for narrow contracts	yes	no	yes
Compatible with graceful shutdown	yes	no	if ignore
Compatible with log-and-continue	yes	no	if ignore
Compatible with throwing defensive checks	yes	no	if ignore
Supports compile-time algorithm selection	no	yes	no

4.2 Syntax and spelling

The `[[throws_nothing]]` annotation fits well with the conventional notion of an attribute: Removing the attribute has no essential effect on a correct program (see P2552R3). Rendering this functionality as a keyword or contextual keyword seems unnecessary.

Putting the `[[throws_nothing]]` attribute in the same location as `noexcept` would seem logical, but for an attribute to appertain to a function, the attribute must occur either before the function declaration or immediately after the function identifier:

```
[[throws_nothing]] void f(int);   // OK
void g [[throws_nothing]] (int);  // OK
void h(int) [[throws_nothing]];   // ERROR: improper attribute placement
```

The original spelling for the attribute was `[[does_not_throw]]`, which happens to have the same number of characters as `[[throws_nothing]]`. The name was changed to `[[throws_nothing]]` to match the "*Throws*: nothing" phrasing that LWG uses when documenting functions that do not throw.

This paper does not propose the ability to make `[[throws_nothing]]` conditional on a compile-time constant Boolean property, like the `noexcept` clause is. Such functionality seems counterintuitive; this proposal deliberately omits any method for querying the `[[throws_nothing]]` attribute, and thus the common idiom of wrapping a function and propagating its `noexcept` property has no equivalent for the `[[throws_nothing]]` attribute. Nevertheless, if conditional functionality is found to be useful (now or in the future), the syntax can be extended with a parameter, i.e., `[[throws_nothing(` *constant-bool-expression* `)]]`.

5 Alternatives Considered

5.1 Switching `noexcept` on and off with a constant expression

One use of `[[does_not_throw]]` is to allow defensive checks to throw an exception through an otherwise-nonthrowing interface. One proposed way to achieve this behavior for nonthrowing functions is to use `noexcept` in such a way that it can be turned off when desired. This approach can be implemented with the help of the preprocessor. For example, using the framework described in "The unit testing universe" section, `noexcept` can be turned off when `CHECKED_MODE` is defined:

```
#ifdef CHECKED_MODE
inline constexpr bool does_not_throw = false;
#else
inline constexpr bool does_not_throw = true;
#endif

void f(int i) noexcept(does_not_throw)  // BAD IDEA!
{
  ASSERT(i < 0);
  // ...
}
```

With this approach, the expression `noexcept(f(0))` will yield different results depending on the `CHECKED_MODE` macro, possibly resulting in different logic paths for debug and release builds, and will thus violate the principle that essential behavior must not be changed by build modes — a principle convincingly advocated for in P2831R0 and P2834R0 and named, by the latter, *Build-Mode Independence*.

6 Effects on the Standard Library

No changes would be needed immediately in the C++23 Standard Library if `[[throws_nothing]]` were adopted. LWG can discuss whether to replace or augment "*Throws*: nothing" in the description with `[[throws_nothing]]` in the interface of functions having narrow contracts that promise not to throw when called in contract.

An immediate change to the C++26 Working Paper might be necessary if any narrow-contract functions targeted for C++26 are currently annotated with `noexcept`; perhaps those annotations should be changed to `[[throws_nothing]]` or perhaps the Standard should omit the annotation and leave it up to the implementation to decide whether to use `[[throws_nothing]]`. Violations of the Lakos Rule already in C++23 could be handled on a case-by-case basis (via DRs). Minimizing such violations would result in greater stability across implementations and versions of the Standard.

7 Implementation Experience

At present, no compilers implement this feature. If this paper receives a favorable response in EWGI, we will implement the proposed facility before presenting it to EWG. Implementation is expected to be a fairly simple delta on the existing implementation of `noexcept`.

8 Formal Wording

Changes are relative to the May 2023 Working Paper, N4950.

Note: This wording is known to be incomplete, Open issues are called out when possible.

Insert the following note somewhere within paragraph 5 **[except.spec]**:

> [*Note*: The `[[throws_nothing]]` attribute is not a non-throwing specification. — *end note*]

Insert the following new paragraph after paragraph 5 **[except.spec]**:

> Whenever an exception is thrown and the search for a handler ([except.handle]) encounters the outermost block of a function previously declared with the `throws_nothing` attribute, it is implementation-defined whether the function `std::terminate` is invoked ([except.terminate]).

Open issue: Does anything need to be said about such a function called indirectly via a function pointer?

Insert the following new subsection at the end of the **[dcl.attr]** section:

> **Throws nothing attribute [dcl.attr.throwsnothing]**
>
> The *attribute-token* `throws_nothing` specifies that a function cannot exit via an exception. No *attribute-argument* clause shall be present. The attribute may be applied to a function or a lambda call operator. The first declaration of a function shall specify the `throws_nothing` attribute if any declaration of that function specifies the `throws_nothing` attribute. If a function is declared with the `throws_nothing` attribute in one translation unit and the same function is declared without the `throws_nothing` attribute in another translation unit, the program is ill-formed, no diagnostic required. The effects of the `throws_nothing` attribute is described in [except.spec].

[*Note 1*: Unlike the exception specification of a function ([except.spec]), whether a function is marked with `throws_nothing` has no effect on the function's type and is not observable through the `noexcept` operator ([expr.unary.noexcept]). — *end note*]

Recommended practice: An implementation should provide to users the ability to translate a program such that all instances of `throws_nothing` result in `std::terminate` being invoked as described above. An implementation should further provide to users the ability to translate a program such that all instances of `throws_nothing` are ignored. The value of a *has-attribute-expression* for the `throws_nothing` attribute should be `0` if, for a given implementation, the `throws_nothing` attribute never causes `std::terminate` to be invoked.

Rationale: The *Recommended Practice* wording is consistent with proposed wording for the Contracts facility; see P2877R0.

[*Example 1*:

```
[[ throws_nothing ]] void f(int x) {
  if (x < 0)
    throw "negative";  // Behavior is implementation-defined if x < 0.
}

static_assert(noexcept(f(-1) == false));  // OK, attribute is not queryable.
```

— *end example*]

9 Conclusion

The `noexcept` specifier is problematic because it can be queried via the `noexcept` operator, which means that it cannot be changed without changing the meaning of a client program. Moreover, the consequence of violating a `noexcept` specification is immediate program termination. By creating similar feature, `[[throws_nothing]]`, that differs only in that it (1) cannot be queried and (2) can be ignored without violating its semantics, we enable optimizing for multiple distinct universes; adopting this proposal achieves the wants and needs of the multiverse.

10 Acknowledgments

Thanks to John Lakos, Joshua Berne, Brian Bi, Mungo Gill, Timur Doumler, and Lori Hughes for reviewing this paper and offering useful improvements. Thanks to Timur Doumler for providing most of the formal wording and Nina Ranns and Joshua Berne for reviewing the wording.