# Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R3** Date: 2019-06-15
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, LEWG

**R3 major changes:** Section 4.3 (regarding OOM) has been substantially revised and expanded. If you've read a previous revision of this paper, you can focus on just that section.

## Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

**§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI.** — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

**§4.2: Programs bugs are not recoverable run-time errors and so should not be reported as exceptions or error codes.** — We must express preconditions, but using a tool other than exceptions. This paper supports the change, already in progress, to migrate `std::` away from throwing exceptions for precondition violations.

**§4.3: Heap exhaustion (OOM) is not like other recoverable run-time errors and should be treated separately.** — We must be able to write OOM-hardened code, but we cannot do it portably by trying to report all failed memory requests. This paper proposes fail-fast for failed allocations of single objects using the default allocator, and reporting failure (via `nullptr` or `bad_alloc`) for failed allocations of arrays using the default allocator or of any allocation using a non-default allocator (such as `new(nothrow)`) that chooses to report OOM accurately.

**§4.5: Some users don't use exceptions because exceptional control flow is invisible.** — We must have automatic propagation, but also the ability to make it visible. This paper proposes allowing `try`-expressions to make exceptional paths *concisely explicit* in calling code, without losing any of the benefits of automatic propagation.

## Contents

# 1 Overview

## 1.1 Prelude: Success-with-info and bugs/corruption are not "errors"

***error:*** *"an act that … fails to achieve what should be done." — [Merriam-Webster]*

*"throwing logic error type exceptions… is not recoverable…" — [Douglas 2018]*

In this paper, "error" means exactly and only "a function couldn't do what it advertised" — its preconditions were met, but it could not achieve its successful-`return` postconditions, and the calling code can recover.

**(1) An alternate result is never an "error" (it is success, so report it using `return`).** This includes "partial success" such as that a buffer was too small for the entire request but was filled to capacity so more can be read on the next call. Consider opening a file: For a general `disk_file` class's constructor that takes a filename, not being able to open the file is a normal outcome (due to file system race conditions) and the type should allow a not-open state; failing to open the file is *not an error*, it does not violate the constructor's postcondition (its class invariant). For an `open_file` class whose invariant guarantees an object of that type represents an *opened* file, or for a higher-level function like `InitializeSubsystem` that depends on a `config.ini` file, not being able to open the file is an error because it prevents achieving that constructor's postcondition (its class invariant).

> **Note** I distinguish "error codes" from "status codes" that could contain non-error results. Conflating "routine" and "error" results in one object makes it harder to keep normal and error processing distinct.

**(2) A programming bug or abstract machine corruption is never an "error" (both are not programmatically recoverable, so report them to a human, by default using fail-fast).** Programming bugs (e.g., out-of-bounds access, null dereference) and abstract machine corruption (e.g., stack overflow) cause a corrupted state that cannot be recovered from programmatically, and so they should never be reported to the calling code as errors that code could somehow handle. For example, these are *not errors*:

- A precondition (e.g., `[[expects...]]`) violation is always a bug in the caller (it shouldn't make the call). Corollary: `std::logic_error` and its derivatives should never be thrown (§4.2), its existence is itself a "logic error"; use contracts, assertions, or similar tools instead.
- A postcondition (e.g., `[[ensures...]]`) violation on "success" return is always a bug in the callee (it shouldn't return success). Violating a `noexcept` declaration is also a form of postcondition violation.
- An assertion (e.g., `[[assert...]]`) failure is always a bug in the function (its code is incorrect).
- Stack exhaustion is always an abstract machine corruption (a function cannot guard against it).

> **Note** For a discussion of heap exhaustion (OOM), see §4.3.

## 1.2 Background and motivation summary

*"… error handling idioms and practices remain contentious and confusing within the C++ community (as well as within the Boost community)." — [Bay 2018]*

C++ is the only major language without a uniform error handling mechanism that is recommendable for all code:

(§2.1) Neither today's exceptions nor error codes are it. Each meets requirements that the other does not.

(§2.2) We are proliferating dual-mode interfaces that try to support *both* models (e.g., `std::filesystem`).

(§2.3) Worse, for reasonably common classes of real-world examples, *neither* is acceptable.

(§2.4) So we keep trying to invent new alternatives (e.g., Expected, Outcome).

One consequence is that "C++" projects commonly turn off exception handling support in all or part of their projects, and are therefore not actually using Standard C++, but using a divergent language dialect with different idioms (e.g., two-phase construction) and either a nonconforming standard library dialect or none at all. This is a deep fracture, and *de facto* a different language: For example, it is not possible to write generic code that works with types written in both styles, or to correctly combine (in some cases, even link) code that uses `std::` features with code that assumes a nonthrowing dialect of the library.

We must make it possible for all C++ projects to at least turn on exception handling support so they can use the standard language and library. So the question is not how to pick a winner from among the many error handling mechanisms; it is how to improve `throw`/`try`/`catch`, the only alternative that supports basic features like constructors, to become a universally usable mechanism, given C++'s constraints of zero-overhead and determinism (both of which today's dynamic exception model violates) and backward source and binary compatibility.

> **Note** Importantly, "zero *overhead*" is not claiming zero *cost* — of course using something always incurs some cost. Rather, C++'s zero-overhead principle has always meant that (a) "if you *don't* use it you don't pay for it" and (b) "if you *do* use it you can't reasonably write it more efficiently by hand."

## 1.3 Design principles

> **Note** These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. — This proposal preserves the clear distinction between normal code and error handling, both when raising an error (`throw` instead of `return`) and handling an error (`catch` instead of normal control flow like `if`/`co_await`). It aims to remove the incentive to resort to compromised designs such as having the same function inconsistently report some errors using an error code and others by throwing an exception. It directly improves consistency by providing a form of exception whose automatic propagation leaks much less type information from otherwise-encapsulated implementation details.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. — This proposal enables better composition by making errors that are propagated across a boundary cleanly composable, including supporting better automatic lossless propagation.
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. — This proposal retains the ability to report all kinds of errors using the same mechanism (with the note that heap exhaustion may be worth treating specially; see §4.3). It aims to evolve exception handling to be a single general way to report errors that is suitable for all C++ code.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

Additional design principles include: Make important things and differences visible. Make unimportant things and differences less visible. — This proposal suggests considering making exceptional control flow visible in source code.

## 1.4 Acknowledgments

Thank you especially to the Direction Group for publishing [P0939R0] as a call for a direction for C++ evolution that explicitly calls out the need to address the ongoing practical problems with exceptions vs. error codes.

Thank you to SG14 and LEWG for their feedback and encouragement, and to the following for reviews and comments on this material: JF Bastien, Charley Bay, Paul Bendixen, Bartosz Bielecki, Vicente Botet, Glenn Brown, Josh Cannon, Jonathan Caves, Alex Christensen, Daniel Colascione, Ben Craig, Pavel Curtis, Guy Davidson, Gabriel Dos Reis, Niall Douglas, Ryan Fairfax, Nat Goodspeed, Chris Guzak, Howard Hinnant, Odin Holmes, Andrew Hunter, Sergey Ignatchenko, Dan Jump, Thomas Köppe, Ben Kuhn, Stephan T. Lavavej, John McCall, Jason McKesson, Jason Merrill, Arun Muralidharan, Phil Nash, Gor Nishanov, Michael Novak, Arthur O'Dwyer, Billy O'Neal, Roger Orr, Sean Parent, Andreas Pokorny, Geoffrey Romer, Ryan Shepherd, Bjarne Stroustrup, Tony Tye, Tony Van Eerd, Ville Voutilainen, Titus Winters, and Michael Wong.

## 1.5 Revision history

**R3:**

- §4.3 on OOM is significantly revised and expanded.

**R1:**

- §4.2 and §4.3 incorporate Rapperswil LEWG poll feedback.

**R0:** Initial revision. Incorporated SG14 telecon discussion and poll feedback.

# 2 Why do something: Problem description, and root causes

## 2.1 Exceptions have not replaced error codes, and vice versa

*"There are still people who argue against all use of exceptions and people who claim that exceptions should be used consistently instead of error codes." — [P0939R0]*

Exceptions are the error handling model that is required by key parts of the language (for constructors and operators) and by the standard library, but are widely banned. This means that a large fraction of the C++ community is not actually using 'real' C++, but are using a language dialect, and either a nonstandard library or none at all.

Even though exceptions are required, and have been available for some 25 years, they have not replaced error codes for error handling in C++. Therefore, they never will unless they are changed in some way to address the reasons they cannot be used universally (see §2.5, "Root causes"). The community are voting with their feet:

- Major coding guidelines ban exceptions, including common modern guidelines endorsed by the world's top advocates of C++ exceptions. For example, the Google C++ Style Guide [GSG] bans exceptions. The Joint Strike Fighter Air Vehicle C++ Coding Standards (JSF++) [JSF++ 2005] was produced by a group that included Bjarne Stroustrup and is published on Stroustrup's personal website, and bans exceptions.
- Many projects ban exceptions. In [SC++F 2018], **52%** of C++ developers reported that exceptions were banned in part or all of their project code — i.e., **most** are not allowed to freely use C++'s primary recommended error handling model that is required to use the C++ standard language and library.
- Committee papers such as [P0829R2] and [P0941R0] embrace standard support for disabling exceptions.
- The C++ Core Guidelines' Guidelines Support Library [GSL] requires exceptions, and cannot be used in such projects. We are already getting requests for a nonthrowing version of GSL, which changes some of its interfaces (e.g., `narrow` reports errors by throwing `narrowing_error` and would have to change).
- Non-throwing dialects of the STL and the rest of the standard library proliferate, and C++ implementation vendors continue to receive requests to support those nonstandard dialects.
- Every C++ compiler supports a mode that disables exception handling (e.g., `-fno-exceptions`).

This is an intolerable rift: Large numbers of "C++" projects are not actually using standard C++.

But switching to error codes isn't the answer either — error codes cannot be used in constructors and operators, are ignored by default, and make it difficult to separate error handling from normal control flow.

## 2.2 Instead, we're actively proliferating dual interfaces that do *both*

*"Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`." — [N3239]*

Because we cannot universally recommend either exceptions or error codes, the community and even the committee are proliferating dual error reporting interfaces that support both, by providing throwing and non-throwing alternatives. Worse, the 'non-throwing' alternatives in the standard are only non-throwing for some kinds of errors, and still also throw to report other errors from the same function.

For example, the C++17 `std::filesystem` library supports reporting *file system errors (only)* as either exceptions or as error codes, often providing a pair of functions, one for each style; both functions still report non-file errors using exceptions. For example, consider `std::filesystem::directory_iterator::operator++`:

```
directory_iterator& operator++();
directory_iterator& increment( std::error_code& ec );    // note: NOT noexcept
```

**Note** `noexcept` was removed from the second function, and a number of similar ones, at the recent Jacksonville meeting. See [LWG 3013] and [LWG 3014]. Before that it was already absent for over a dozen similar functions per the policy summarized in the next paragraph.

The current design policy for `filesystem` is that, for file system codes only (which can also be just status codes), the first function of each such pair reports them using exceptions and the second reports them using `error_code`; both alternative functions can still throw exceptions for other non-file errors. This means that inside this dual error reporting design (two alternative functions) is a second nested dual error reporting design (in the same function, some errors are reported via exceptions and others via `error_codes`), and this is intentional.

**Notes** This has surprised a lot of people, including SG14 in [P0824R1] sections 4.2 and 4.9. I find that programmers who encounter the `filesystem` API make the assumption that the second alternative is for programs that don't want to throw exceptions or that don't have exception handling turned on. So it is important to teach all `filesystem` users that `filesystem` does not actually generally support a non-throwing mode, despite the overloads that appear to do so. — Rather, the motivation to not throw exceptions appears to be more because "routine" status is reported using the same code type that also communicates true errors (see §1.1 point (1) and accompanying Note, in this paper), and so callers that use exceptions exclusively were found to be often littered with local `try`/`catch` blocks to handle "routine" events. A different design approach for libraries like `filesystem` would be to distinguish "routine"/informational status codes as distinct (a separate object) from error codes, and report only the latter as errors.

We are following this dual design policy even though we know it has serious drawbacks:

- **(worst) Makes error handling harder.** It's hard enough to get call-site programmers to perform consistent and correct error handling when given a single consistent method of error reporting. Now we're giving them two alternatives to choose from — and then in one of those alternatives additionally reporting errors in two ways from the same function, asking callers to write two error handling twice using different styles. This makes it difficult for callers to write reliable code.
- **Interface bloat.** It bloats the library's interface, which includes both the library's documentation (external) and an implementation's test burden (internal).
- **Encourages dialects (I).** It actively encourages C++ dialects, because some callers use exceptions and some use error codes, and both are actively supported by the API.
- **Inconsistency.** It eliminates the ability to use a consistent function name at least for operators since these cannot be overloaded in this way (e.g., `operator++` vs. `increment`, above).
- **Encourages dialects (II): Discourages other language features.** It creates a broader ripple effect through the language by adding a reason to avoid otherwise-recommended unrelated features (e.g., C++ overloaded operators).

Despite all these drawbacks, within the C++ committee we are now having active discussions, not about solving the underlying problem so we can stop creating dual interfaces, but instead about applying this pattern to still more parts of the standard library (e.g., networking). The above example of `directory_iterator::operator++` also acknowledges implicitly that even the standards committee agrees that exceptions are not considered sufficient even in the cases where they have the strongest possible language advantage over error codes, namely for operators and constructors. No other language I know of has such a bifurcation problem.

## 2.3 Worse, for some real-world code *neither* is acceptable

*"Recent threads on the Boost email-list continue to highlight the ongoing confusion and disagreement even over the proper or idiomatic use of `std::error_code` … One might think such discussions should by now be resolved; but no…" — [Bay 2018]*

*"On table-based exception handling implementations… A `throw...catch` cycle is always at least thousands of times more expensive than a `return` statement, and always must be so, even when the `throw...catch` is inlined" — [Douglas 2018]*

In some real-world code, neither an exception nor an error code is acceptable. A poster child example is a constructor or overloaded operator that can fail, and must be usable in memory-constrained and/or real-time code:

- **It cannot use exceptions**, because the space and time cost of today's dynamic exception handling is nondeterministic and so cannot be guaranteed to fit in bounded space or bounded time. This is why exceptions are banned in JSF++ [JSF++ 2005] and the Mars Rover flight software [Maimone 2014].
- **It cannot use error codes.** For constructors, using error codes means embracing a poor and incompatible C++ dialect, either pervasively using two-phase construction and "is-constructed" tests on every type with a fallible constructor (for example, see Outcome 2.0: Result returning constructors) or replacing constructors with factory functions. For operators, using error codes means not using operators at all but replacing them with named functions (for example, see the preceding `std::filesystem` example which renames `operator++` to `increment`).

Yet the standard library itself, including STL, specifies constructors and operators that can fail. So we cannot easily use a conforming standard library in memory-constrained and/or real-time code; that would require modifying it to report errors in another way (and in an incompatible dialect of C++, per above), or leaving its design as-is but applying the hammer of disabling exceptions and just ignoring errors (unacceptable in general).

## 2.4 And we're proliferating new patches and alternatives

*"Note that `expected` can also act as a bridge between an exception-oriented code and a nothrow world." — [P0323R3]*

*"Exception throwing is absolutely at the heart of Outcome. That's why Outcome != Expected" — N. Douglas, quoted in [Bay 2018]*

**Proliferation of patches to make error codes better.** We have ongoing active discussions, such as in SG14, about "exception-less error handling" using C++11 `std::error_code` or an evolution thereof (see [P0824R1]). Also, C++17 added the `nodiscard` attribute for "important" return values, motivated in part by returned status information that should not be ignored (see [P0068R0] example 4). Note that `nodiscard` is broadly useful and desirable; however, the specific use of relying on it to make sure callers don't silently ignore errors is a "patch" in terms of the error handling model.

**Proliferation of new library-only solution attempts.** The C++ committee and community continue to consider new alternatives in new standardization. For example:

- In the committee, we are advancing the proposed `std::experimental::expected<SuccessResult,ErrorResult>` [P0323R3]. As noted in the paper: *"C++ already supports exceptions and error codes, `expected` would be a third kind of error handling."*
- Boost, while aware of this, continues to pursue evolving a distinct `outcome::result<SuccessResult>` with different tradeoffs, notably lower run-time overhead than expected for `expected`. From the 2018 Boost acceptance report for Outcome v2 [Bay 2018], emphasis original: *"The prime motivation for acceptance is: Reviewers have real-world use cases **today** for which they found Outcome to be an effective and best available alternative; and which is consistent with current-need and expectations; and which is consistent with ongoing C++ Standard evolution efforts. From the Library Author: 'Outcome is really an abstraction layer for setting per-namespace rules for when to throw exceptions. Exception throwing is absolutely at the heart of Outcome. That's why Outcome != Expected, and why it ICEs older compilers, and why C++ 14 is needed.' "*

As library solutions without language support, these approaches have two major problems: First, they are fundamentally attempts to regain use of the return value for error reporting, and by fusing "success" and "error" returns they force callers to perform error handling using only normal control flow constructs to inspect a merged value. Second, they contribute to fracturing C++ error handling because they are adding a third or a fourth style; for example, in their current form, it is not clear whether these would be universally adoptable throughout `std::filesystem` to resolve its dual-mode problem, and [Bay 2018] includes the note that Outcome is not intended to be usable for all error handling.

The good news is that these efforts are blazing trails, and converging, in a good direction: They are already very close to expressing a library type that is suitable for universal C++ error reporting, with strong efficiency and fidelity characteristics. That's important, because it means we may now be at a point where the library type is sufficiently baked for the language to be able to embrace it and help them (this proposal).

**Notes** There are two families of use cases given for `expected`, and only one is about error handling: (1) `expected<T1,T2>` where both paths are normal "routine" control flow, and `T2` is an alternate result for a "routine" outcome; for this, the authors of `expected` acknowledge that `variant<T1,T2>` might be a more natural choice. (2) `expected<T,E>` where `E` really represents an error; for this, I think there is real benefit in this paper's proposed language support to keep the error-handling paths distinct and automatically propagate the errors.

I'm not actually against having ValueOrError types like `expected` and `outcome`; the problem is just that it diverges the programming model for calling code, which always interferes with generic code in particular; for example, I cannot write a template that can equally invokes two types or functions, one of which reports errors using exceptions and the other using `expected`. Fortunately, and importantly, it may be possible to unify these using `try`-expressions; see §4.5.1, particularly Jason McKesson's observation in the Note.

## 2.5 Root causes: Why we can't just use exceptions everywhere today

*"I can't recommend exceptions for hard real time; doing so is a research problem, which I expect to be solved within the decade" — [Stroustrup 2004]*

Above, we enumerated the performance issues with today's dynamic exception handling model: binary image bloat, run-time cost, and deterministic run-time space and time cost (when throwing).

The root cause of these problems is that today's dynamic exception handling model violates two of C++'s core principles, zero-overhead and determinism, because it requires:

- throwing objects of *dynamic types*, which requires **dynamic allocation to throw** and **dynamic RTTI to catch by type**; and
- using *non-local by-reference* propagation and handling semantics, which requires non-local coordination and overheads, and requires arbitrarily many exceptions with distinct addresses at the same time.

For additional details beyond what is covered below, see section 5.4 of the Performance TR, [ISO 18015:2004].

**(1) Today's exception handling is not zero-overhead (binary image size, run-time space and time).** Exception handling is one of two C++ language features that violates the zero-overhead principle, that "you don't pay for what you don't use" and that "when you do use it you can't reasonably write it more efficiently by hand." For example, just turning on exception handling support in a project previously compiled without exception support — i.e., one that is not yet throwing *any* exceptions at all — commonly incurs significant binary space overhead; I regularly hear +15% reported (Chris Guzak in personal communication regarding Windows internal examples, and +16% reported by Ben Craig on the SG14 mailing list for a different code base and environment), and I have recently seen other Windows internal examples with +38% bloat, down from +52% after recent additional back-end optimization (Ryan Shepherd, personal communication). The overhead arises in various places: In the binary image, we have to store jump tables or other data/logic. At run time, most implementations reserve additional stack space per thread (e.g., a 1K reservation, to save a dynamic allocation) and require and use more-expensive thread-local storage.

**(2) Today's dynamic exception handling is not deterministic (run-time space and time cannot be statically bounded), because `throw` requires dynamic allocation and `catch` of a type requires RTTI.** This is the primary reason exceptions are banned in many real-time and/or safety-critical environments (for example, many games, coding standards like JSF++ [JSF++ 2005], and environments like the Mars Rover flight software [Maimone 2014]). C++ allows there to be multiple active exception objects of arbitrary types, which must have unique addresses and cannot be folded; and it requires using RTTI to match handlers at run time, which has statically unpredictable cost on all major implementations and can depend on what else is linked into the whole program.[1] Therefore during stack unwinding the exception handling space and time cost is not predictable as it is with error codes. Adequate tools do not exist to statically calculate upper bounds on the actual costs of throwing an exception.

### 2.5.1 Examples of inherent overheads

Here are some specific examples of required overheads.

Note that all of the overhead examples in this subsection are inherent in the model of "throwing dynamic types using non-local by-reference propagation" — the costs cannot in general be avoided simply by using a smarter implementation strategy (they can only be moved around, such as by using table-based vs. frame-based implementations, or by using heap vs. pin-the-dead-stack allocation), and they cannot in general be optimized away (even with heroic potential optimization efforts that implementations do not actually attempt today).

> **Note** There have been extended arguments about whether the choice of table-based vs. frame-based exception handling implementation strategies might be the reason why exceptions have not been universally adoptable. It isn't. For details, see section 5.4 of the Performance TR, [ISO 18015:2004]. —

[1] Absent heroic optimizations, such as fully inlining all functions called from a `catch` block to prove there is no re-`throw`.

Briefly: Table-based implementations are better when failure almost never happens, and frame-based shines when failure is common, but both still incur non-local costs just to enable exceptions regardless of whether, or how often, they are thrown, and both incur some cost even on the success path. Neither implementation style can achieve zero-overhead or determinism, because the costs are inherent in exceptions' demands for additional binary image code/data, run-time heap allocation, and dynamic run-time typing — table-based vs. frame-based is just moving those costs around, not eliminating them.

**(1) Today's dynamic exceptions can require arbitrarily many exceptions in flight with unique addresses.** Handling an exception can cause additional exceptions (of potentially unrelated dynamic types) to be thrown from the `catch` handler before the exception being handled can be destroyed. Multiple exceptions in flight cannot be folded using normal optimizations for variable folding, and so because arbitrarily many exceptions can be in flight, and their number is not in general statically predictable, throwing an exception requires arbitrary amounts of memory.

**(2) Today's dynamic exception objects cannot be allocated normally in the local stack frame.** This leads to unpredictable time space and/or time costs in various ways. Here are two typical examples:

- On platforms that use the [Itanium ABI], exceptions are required to be allocated on the heap (modulo potential optimizations that are not actually implemented today, such as the proposed LLVM optimization in [Glisse 2013]). Heap allocation requires unpredictable time, even on allocators that avoid global synchronization in the memory allocator.
- On Windows platforms, exceptions are technically allocated on the stack, but they are far from normal stack allocations: When an exception is thrown, the stack contents are destroyed by unwinding back to the `catch` handler, but the now-unused stack space itself is not yet deallocated until the handler ends — in effect the stack storage is "pinned" until the original exception can be destroyed. This means that the `catch` handler code must run at a stack location *beyond* the stack depth where the being-handled exception was thrown, skipping the dead space — and this repeats recursively for any additional exceptions thrown during handling, and C++ today allows arbitrarily many such exceptions to be created (see previous point). For a simple example of just three such in-flight exceptions and how they multiply stack usage, see the Appendix. In this implementation strategy, the stack memory usage is therefore a total of the individual stack depths of each path that threw an exception while another exception was already active, and I do not know of tools that compute a static memory use bound. (I have not tried to measure whether this attempt at 'in-stack-memory-but-not-really-stacklike' allocation is typically better or worse overall than just doing a real heap allocation; it will nearly always be worse in total memory consumed, but it does avoid contention on the global allocator.)

**(3) Therefore, today's dynamic exceptions cannot share the return channel.** When an exception is thrown, the normal return channel is entirely wasted. That itself is an inherent architectural pessimization.

**(4) Today's dynamic exceptions require using some form of RTTI to match handlers.** The cost of RTTI is generally nondeterministic in both space and time.

**Note** RTTI is the other C++ language feature that violates the zero-overhead principle; exceptions and RTTI are so widely disabled that [P0941R0] proposes special feature test macros for testing the absence of only those C++98 language features. We need to fix RTTI too, but this is not that paper. However, because exceptions rely on RTTI (by propagating and manipulating dynamically typed ex-

ceptions) so that the cost of RTTI is indirectly part of the exception handling cost, here is a brief summary of why RTTI violates zero-overhead and the two issues that most directly affect exception handling: [2]

> First, it requires support for `typeid`, including `typeid.name()`, which is effectively metadata. Normally C++'s zero-overhead design rejects "pay for what you don't use" overheads that add space or time cost even when not used; the usual poster child examples are "always-on" or "default-on" (a) metadata (e.g., we have always rejected storing even the names of enumerators) and (b) garbage collection (e.g., we support it via opt-in libraries but not as the global default). The one place C++ adds required metadata is in `typeid`, especially `typeid.name()`.
>
> Second, it does not distinguish dynamic casts that have different costs. For example, the following have different power and different costs (and are already distinguished in the [Itanium ABI]: (a) downcast to the statically unknown most-derived type (complete object); (b) downcast to a statically known derived type (not necessarily most derived); (c) cross-cast to a statically known sibling type; and (d) upcast from the statically unknown most-derived type to a public statically known base type. Because `dynamic_cast` must perform all of the first three operations (a) (b) and (c), it is necessarily at least as expensive as the most expensive of all three. (See [O'Dwyer 2017] for a lucid treatment.) Exception handling only needs (d).
>
> To fix the above two issues (in the context of enabling a more efficient dynamic exception handling implementation), we could provide a version of RTTI for `catch` implementation use only that is not disabled when RTTI is otherwise disabled, and that does not include `typeid` support and includes support only for dynamic casting of type (d), with the caveat that (d) might still violate either the zero-overhead principle (either by generating additional static data in the vtable to enable constant-time casting as demonstrated in [O'Dwyer 2017] slides 40-42, or by avoiding additional static data at the cost of non-constant-time casting which would leave it unsuitable for real-time code).
>
> Third (and this might or might not be able to be mitigated by the approach in the previous paragraph), the cost of RTTI can be effectively unpredictable because linking in unknown third-party shared libraries can dramatically affect the performance of RTTI lookup, and thus the performance of exception handling. In general we cannot predict whether some end user, or even customer of that end user, will not combine our code with some other code in the same process; Niall Douglas reports real-world cases where a user's linking in other code caused the cost of `throw…catch` to rise dramatically (e.g., 500ms on a heavily loaded machine) due to the environment-specific unpredictability of the RTTI cost.

For the above reasons, major projects and guides (e.g., Firefox, Chrome, the Google C++ Style Guide [GSG]) actively discourage or ban using RTTI and `dynamic_cast`. This usually means that these projects cannot use exceptions either, because today exceptions rely on RTTI.

The projects work around their lack of `dynamic_cast` by using `static_cast` downcasts, using a visitor pattern, or rolling their own homegrown dynamic casting method (e.g., storing a type tag for a

[2] There are other issues less directly relevant to exception handling. For example, in addition to these overheads, some implementations of `dynamic_cast` incur needless extra run-time inefficiencies, such as by performing textual string comparison as part of the cast operation. Those overheads can be fixed to incrementally improve RTTI performance, but those fixes are not germane here because they don't materially change the RTTI impact on exception handling.

known class hierarchy, which does not scale universally). This continues to cause new C++ code security exploits due to type confusion vulnerabilities, where the root cause analysis of many recent security incidents has observed that the code should have used `dynamic_cast`, but did not because of its binary image space and/or run-time costs (for example, see [Lee 2015], paragraphs 2 and 3).

It is an open research question whether C++'s currently specified RTTI is implementable in a way that guarantees deterministic space and time cost. [Gibbs 2005] describes an approach to get constant-time dynamic casting in constrained class hierarchies by having the linker assign type identifiers, but it does not support dynamic libraries or hierarchies or arbitrary shape and size, and so is not a general solution. The two known followup papers [Dechev 2008] and [Dechev 2008a] did not attempt to address those issues, but focused on contributing incremental improvements to the heuristic for generating type identifiers.

See also:

- §4.6.1: "Wouldn't it be better to try to make today's dynamic exception handling more efficient, *instead of* pursuing a different model?"
- §4.6.2: "But isn't it true that (a) dynamic exceptions are optimizable, and (b) there are known optimizations that just aren't being implemented?"

Fortunately, having exception handling with automatic propagation does not require a model with these properties. We have existing counterexamples: For example, although today's C++ dynamic exception handling is not isomorphic to error codes, Midori's [Duffy 2016] and CLU's [Liskov 1979] exception handling models are isomorphic to error codes which enables more efficient implementations, and does not preclude automatic propagation.

# 3 What we want: Ideal error handling characteristics

## 3.1 Summary of the ideal: We need exceptions' programming model

*"Conflating error handling and control flow is a crime against readability and conciseness."*
*— Michael Novak, personal communication*

This section lays out what I believe are ideal error handling characteristics. They are not unique to C++; I believe they apply to most modern languages.

| Ideal | Exceptions | Error codes | `expected<T,E>` | `outcome<T>` |
|---|---|---|---|---|
| **A. "Error" flow is distinct from "success"** | | | | |
| When raising (distinct from normal `return`) | Yes (`throw`) | No | Partial (`return unexpected`) | Yes (`return success` vs. `return failure`) |
| When handling (distinct from success code) | Yes (`catch`) | No | Partial (`.value()` throws) | Partial (policy determined) |
| **B. Error propagation and handling** | | | | |
| Errors can only be ignored explicitly (not ignored silently by default) | Yes | Partial (`nodiscard`, warnings) | No (in current proposal) | Partial (policy configurable) |
| Unhandled error propagation is automated | Yes | No | No | No |
| Unhandled error propagation is visible | No (Yes if §4.5) | Yes | Yes | Yes |
| Writing an error-preserving error-neutral function is simple | Yes | No | ? | Yes |
| **C. Zero-overhead and determinism** | | | | |
| Stack allocated (no heap) | No (Yes if §4.1) | Yes | Yes | Yes |
| Statically typed (no RTTI) | No (Yes if §4.1) | Yes | Yes | Yes |
| Space/time cost equal to `return` | No (Yes if §4.1) | Yes | Yes | Yes |
| Space/time cost fully deterministic | No (Yes if §4.1) | Yes | Yes | Yes |

**Note** This paper does not address other potential improvements that would require a source breaking change, such as that function declarations should default to "does not fail." In the future, I hope to bring proposals to address those issues in the broader context of exploring how to take a source breaking change that could change defaults and in other ways enable further C++ simplification, but they are beyond the scope of this paper.

Group A: "Normal" vs. "error" is a fundamental semantic distinction, and probably the most important distinction in any programming language even though this is commonly underappreciated. Therefore, the distinction should be surfaced explicitly (though as elegantly as possible) in language syntax and program structure.

Group B: True errors (as opposed to partial-success or other success-with-info) are important and should be handled even if by explicitly doing nothing. Any approach that allows them to be silently ignored will incur long-term cost to program robustness and security, and to a language's reputation. Further, they should be propagated in a way that the programmer can reason about. — The one place that exception handling fails the ideals shown here is that exception propagation between the `throw` site and the `catch` handler is invisible in source

code, which makes exception-neutral code (which predominates) harder to reason about and is primarily addressed by widespread use of RAII stack-based variables (which are good for many reasons besides exception safety).

Group C: This group is "because this is C++," but it's also where exception handling most falls short today. The proposal in §4.1 is motivated by the observation that the costs are associated with being able to throw arbitrarily typed exceptions.

## 3.2 Goals and non-goals

This paper aims at two essential goals, that we must achieve to keep C++ unified (whether via this proposal or in some other way).

**(1) We must remove all technical reasons for a C++ project to disable exception handling (e.g., by compiler switch) or ban use of exceptions, in all or part of their project.** This does not mean requiring a project to actually use exceptions for all their error reporting. It just means that every C++ project be able to use the standard C++ language and a conforming standard library.

| | |
|---|---|
| **SG Poll** | The 2018-04-11 SG14 telecon took a poll on whether the above is a problem worth trying to solve: Unanimous consent. |

**(2) We must reduce divergence among error reporting styles.** This means converging as many of the divergent error reporting styles as possible by providing a usable model that can subsume some of the others.

Non-goals (but we might effectively reach them anyway, at least in part):

- It is not a goal to make exceptions safe for propagation through C code. — However, because this proposal defines a kind of exception that is implemented as an error return, I believe this proposal could make it possible for C and other-language code to correctly invoke C++ functions that use the proposed exception model to report errors and that otherwise are written in the C subset.
- It is not a goal to enable errors to be handled using normal control flow constructs. — However, §4.5 describes how this proposal puts us on a path where programmers can write code in exactly the same style as using `expected<T,U>` today, but with the improvement of keeping the normal and error paths as fully distinct (`catch` instead of using normal control constructs).
- It is not a goal to enable distantly-handled errors to contain arbitrary *programmatically*-usable information. Distantly-handled error details primarily need to be *human*-usable (e.g., debugging and trace logging), and a `.what()` string is sufficient.

# 4 How to get there

*… The [Outcome] Library Author can be congratulated (or scolded) for exploring work or attempting Boost community review in such a contentious space.” — [Bay 2018]*

This section proposes a solution — not without trepidation, because I understand this touches an electrified rail.

Error handling is perhaps the most fundamental design point of any programming language. It cannot be changed lightly. However, if there are chronic unresolved issues with error handling, then addressing those successfully can have outsized leverage to deliver broad improvement across all uses of the language — if we can design for backward source and binary compatibility, so that new and old code can interoperate seamlessly.

## 4.1 Core proposal: `throws` values (addresses §3.1 groups C and D)

| **SG Poll** | The 2018-04-11 SG14 telecon took a poll on pursuing this direction: 12-2-0 (Favor-Neutral-Against). |
|---|---|

### 4.1.1 Elevator pitch

This proposal aims to marry the best of exceptions and error codes: to allow a function to declare that it *throws values of a statically known type*, which can then be implemented exactly as efficiently as a return value.

Throwing such values behaves as-if the function returned `union{R;E;}+bool` where on success the function returns the normal return value `R` and on error the function returns the error value type `E`, both in the same return channel including using the same registers. The discriminant can use an unused CPU flag or a register.

The entire implementation of throwing and propagating such exceptions is entirely local within a function and its stack frame (no need for separate tables, no separate allocation outside the stack frame), is statically typed (no need for RTTI), and is equally deterministic in space and time as returning an error code. It is at most zero overhead compared to returning an error code, and can be negative overhead in practice compared to returning an error via an `error_code&` out-parameter because an out-parameter cannot share the return channel.

Expanding the elevator pitch to specific audiences:

- **If you love exceptions, including you wish you could use exceptions but can’t tolerate their cost**: This is exception handling, with error handling separated from normal control flow and automatic propagation and never-silently-ignorable errors — plus the special sauce that if you agree to throw an `error` value you get a more efficient implementation that is truly zero-overhead and fully deterministic in space and time.
- **If you love `expected`/`outcome`**: This is embracing `expected`/`outcome` and baking them into the language, the function always returns exactly one of `R` or `E` — plus the special sauce that you get automatic propagation so you don’t have to manually return-up the results, and with a distinct language-supported error path so that callees can write `throws` (instead of `return unexpected`) and callers get to cleanly put all their error handling code in distinct `catch` blocks (instead of `if(!e)` blocks) while still writing in the same basic `expected` style (see §4.5).
- **If you love error codes:** This is just giving a function two return paths, one for success and one for failure where the latter returns an error code as usual — plus the special sauce that the language lets you distinguish the two, the error code doesn’t monopolize your natural return value channel, you don’t have to propagate the error by hand, and you can’t forget to check errors.
- **If your project needs fail-fast on all heap exhaustion:** See §4.3 §4.4.

### 4.1.2 `std::error` type

*"By allowing multi-level propagation of exceptions, C++ loses one aspect of static checking. One cannot simply look at a function to determine which exceptions it may throw." — [Stroustrup 1994] p. 395*

Let *relocatable* mean movable with the semantics that the destructor of the moved-from object is never called.

Let *trivially relocatable* mean that the move step is trivial (but the destructor need not be trivial).

**Notes** "Trivially relocatable" implies that, given two objects `src` and `dst` of type `T`, performing a move from `src` to `dst` followed by performing destruction `src.~T()` is functionally equivalent to just copying the bytes from the source object to the destination object. A roughly-equivalent formulation is that moving `src` to `dst` is functionally equivalent to just copying the bytes from `src` to `dst` and then copying the bytes from a default-constructed `T{}` to `src`.

Any trivially copyable type is also trivially relocatable, but many types are trivially relocatable without being trivially copyable, including (in most implementations) `unique_ptr`, `exception_ptr`, and `string`.

See also the directly related `[[move_relocates]]` proposal [P1029R0]. If that proposal is adopted, `std::error` can be annotated using the general `[[move_relocates]]` mechanism. In the meantime, for this paper I define the term only in order to define `error` itself as a type having trivially relocatable semantics, and to define the destructor treatment of a user-selectable error type `E` in §4.6.5 if it is/isn't relocatable.

There have been suggestions for such a general language feature, under names such as "destructive move," but neither this proposal nor [P1029R0] proposes that.

[O'Dwyer 2018a] demonstrates `is_trivially_relocatable` as an opt-in library tag, where making the libc++ implementation of `vector<unique_ptr<int>>` relocation-aware, and tagging `unique_ptr` as relocatable, improved `.reserve()` reallocation performance by 3×.

See this Godbolt example provided by Niall Douglas which demonstrates that having either a trivial move constructor or a trivial destructor is sufficient to return `error` in registers on the Itanium ABI. Using the related new (2018-02) Clang extension `[[clang::trivial_abi]]` (see [O'Dwyer 2018b]) it is possible to get register-passing capability for a wider variety of RAII types; see this Godbolt example provided by Arthur O'Dwyer.

The single concrete type `error` is an evolution of `std::error_code`; see also related paper [P1028R0]. It has the following ideal requirements, including the improvements suggested by SG14's review in [P0824R1]:[3]

- It always represents a failure (there is no `0` success value). A default constructor would construct a general or "other" nonspecific error value.
- Its size is no greater than two pointers, typically a "payload" (usually an integer) plus a `constexpr` "domain" (usually a pointer or hash value that is used only for its type to distinguish the domain).

[3] See [Douglas 2018c] for a sample prototype implementation, which claims to meet all of the requirements stated in this list. It is a refinement of `system_code` (an alias for `status_code<erased<intptr_t>>`) from [Douglas 2018a], which itself is just starting to be brought to Boost and SG14.

- Its "domain" discriminant (similar to `std::error_category` but with the improvements suggested in [P0824R1]) is able to represent all causes of failure in the C++ standard library, as well as POSIX system codes, Windows `NTSTATUS`es, COM `HRESULT`s, and other popular error reporting systems.
- It is type-erased, allocation-free, trivially relocatable, constant-time in all operations, ABI-safe, and safe to use in header-only libraries, while also non-lossy to preserve the original cause of failure.
- It provides `weak_equality` heterogeneous comparison that performs *semantic equivalence* comparison across domains, which aids composability; for example, "host unreachable" errors from different domains (e.g., Win32 and POSIX) compare equivalent to each other and to `errc::host_unreachable` which can be queried in portable code without being dependent on the platform-specific source error.

**Note** This proposed `std::error` is a library type. As usual for a common standard low-level type it can have implementation-defined/compiler-supported "gravy," as we do with the `std::` comparison categories. In this case, that can include making it work even better with C implementations, as there is a coordinated proposal going through the C committee at the same time as this proposal for C++. All of this does not affect the library specification of the type, and is in addition to, not instead of, working great for C++.

### 4.1.3 `throws` static-exception-specification

This paper proposes that a function (including lambda function) may declare a **static-exception-specification** of just `throws` to indicate that the function can fail. If the function fails, it throws an object of type `std::error` implemented as-if returning it as an alternate return value (i.e., on the stack).

For example:

```
string f() throws {
    if (flip_a_coin()) throw arithmetic_error::something;
    return "xyzzy"s + "plover";              // any dynamic exception is translated to error
}

string g() throws { return f() + "plugh"; }  // any dynamic exception is translated to error

int main() {

    try {
        auto result = g();
        cout << "success, result is: " << result;
    }
    catch(error err) {                       // catch by value is fine
        cout << "failed, error is: " << err.error();
    }

}
```

**Note** I considered using `throw`, but we also want a "does this throw static exceptions" operator analogous to the `noexcept` operator (see §4.1.4), and we can't use `throw` unambiguously for that operator. So for consistency between this declaration and the operator, I am using the strawman syntax `throws`. Using `throws` also helps to avoid any confusion with the mostly-removed dynamic exception specification `throw(/*...*/)` syntax.

For a function `f` declared with a static-exception-specification `throws`:

- All declarations of `f` must be declared `throws`, including in base classes if `f` is a virtual override.

- `f` behaves as-if `noexcept(false)` when queried by the `noexcept` operator and the `*noexcept*` and `*nothrow*` traits (e.g., `move_if_noexcept`, `is_nothrow_move_constructible`).
- Conceptually, in the case of failure `f` behaves as-if it were declared with a return type of `error`. The normal and error returns share the same data channel and exactly one is used.

**Notes** This includes being able to return `error` in registers. There are no functions declared with `throws` today, so we have an opportunity to define the ABI for this new set of functions, as a new case that extends existing calling conventions. For example, we can expand the number of registers (e.g., to 4 or 6 on x64, to 8 on AArch64), and use one of the unused CPU flag bits to indicate whether those registers contain a value or an error.

An alternative would be to formally specify this be implemented as an `E*` "out" parameter, so that if the function is otherwise callable from C (or other languages that understand C as de facto lingua franca) then the error handling is consistently available from calling code in those languages. An out-parameter implementation strategy could generate more compact code for exception-neutral code, and reduce total stack usage. — We will prototype and measure both alternative implementations.

- For any throw-expression in `f`'s body that has no argument (i.e., re-`throw;`): It must appear in a `catch` block and behaves as-if `throw e;` where `e` is the catch block parameter.

**Note** Alternatively, for an `error` value only, we could disallow anonymous re-`throw` and require `throw e;`. But then we would still want to support anonymous re-`throw` as a synonym in migration/compatibility mode (see §?).

- For any throw-expression in `f`'s body that has argument, `throw` *`expr`*`;`, where *`expr`* is convertible to `error`:
  - If `f` is in a block with a surrounding local `catch(error)` or `catch(...)` handler, control goes to that handler as-if via a forward `goto`.
  - Otherwise, it behaves as-if `return` *`expr`*`;`.

**Notes** The usual rules apply, such as that if *`expr`*'s or `e`'s type is not `error` or convertible to `error` then throw-expression is ill-formed.

This specification is deliberately in terms of forward-`goto` semantics (zero overhead by construction), not in terms of a notional try-catch where we then rely on optimizers to elide the overhead (attempting to claw back zero overhead by optimization).

- When calling another function `f2` that also has a static-exception-specification and that throws an exception `e`, the effect is as-if `throw e;`.
- When one of today's dynamic exceptions is unhandled in `f`'s body, regardless of whether it originated from a nested function call or a `throw` statement throwing a dynamic exception, the exception is automatically caught and propagated: If the caught exception is of type `error`, we just return it. Otherwise, it is translated to an `error` with a meaningful value for all `std::` exception types; for example, `bad_alloc` would be translated to `std::errc::ENOMEM`. Otherwise, we can additionally store as payload a raw pointer to an `exception_ptr` to the dynamic exception (see §4.6.4), without sacrificing trivial movability.

**Notes** `current_exception` and `exception_ptr` (and their possible use of TLS) are not needed for `error`, because an `error` exception has a static type and is used by value. `current_exception` and `exception_ptr` are intended to be used only for today's dynamic exceptions, including to wrap them into an `error`.

The mapping from exception types and values to `error` values will need to be fully specified in the standard.

We should include a customization point to allow enabling automatic translation also for other exception types.

- If `f` is a virtual function, then: Every base function that `f` overrides must be declared `throws`. Every further-derived override of `f` must be declared either `throws` or `noexcept`.

**Note** If a base function is declared `noexcept`, a derived override must also be declared `noexcept`.

For a function declared without a static-exception-specification:

- When calling another function `f` that has a static-exception-specification and that throws an exception `e`: If the `error` is a wrapped `exception_ptr`, it rethrows the dynamic exception. Otherwise, if the `error` value corresponds to one of the meaningful values for a `std::` exception type, it throws an exception of that type; for example, `std::errc:ENOMEM` would be translated to `bad_alloc`. Otherwise, the effect is as-if `throw e;`, that is, it throws the `error` itself as a dynamic exception.

**Notes** Today, implementers are permitted, but not required, to make `exception_ptr` trivially relocatable. If it is, `error` can hold an `exception_ptr` directly as its payload. If it is not, the `exception_ptr` can be allocated on the heap and `error` can hold a raw pointer to it and destroy it when it's done with it (at the end of the `catch` handler that consumes it without rethrowing).

Some reviewers have expressed the opinion that it might be better to require code to manually translate between static and dynamic exceptions. The main motivation for automating this is two-fold: (1) We want to make it as easy as possible to upgrade existing non-`noexcept(true)` functions by just adding `throws`. If we don't do this, then the programmer still has to write a `try`/`catch` by hand. (2) The programmer can't write better code than we could by automatically translating the exception. So, since the `try`/`catch` is both always necessary and cannot be written more efficiently by hand, it should be default and automatic. — That said, tools could warn when such implicit translation is happening, such as to find not-yet-upgraded places in an existing code base.

### 4.1.4 Conditional `throws` and operator `throws`

A static-exception-specification of `throws(`*`cond`*`)` has the basic meaning `noexcept(!`*`cond`*`)` but additionally can distinguish between static (default) and dynamic exception reporting. For example:

```
template<class T>
struct X {
    void X(X&&) throws(!is_nothrow_move_constructible_v<T,T>);
    X& operator=(X&&) throws(!is_nothrow_move_assignable_v<T,T>);
};
```

The condition *cond* evaluates to a value of type `enum except_t { no_except=false/*0*/, static_except=true/*1*/, dynamic_except/*=2*/ };`.

The operator `throws(expr)` performs a compile-time computation returning the `except_t` used by *expr*, and returns `no_except` if the *expr* is declared to be `noexcept`, otherwise `static_except` if *expr* is declared to be `throws`, otherwise `dynamic_except`.

> **Note** We can't use `throw` unambiguously for the operator, hence `throws`. This is the primary motivation for using `throws`, not `throw`, as the keyword for declaring that a function uses static exceptions.

This permits an algorithm such as `std::transform` to efficiently be `noexcept`, report errors using static exceptions, or report errors using dynamic exceptions exactly as the caller-provided operation does:

```
template< class In, class Out, class Op >
Out transform( In first, In last, Out out, Op op ) throws(throws(op(*first)))
```

In this example, each instantiation of `transform` reports error however `op` does, using exactly one of static exceptions, dynamic exceptions, or `noexcept`.

> **Note** A function that wants to adapt to multiple suboperations that could have different error modes (e.g., some could be `no_except`, others `static_except`, and/or still others `dynamic_except`) can compute how it wants to report errors. It is expected that a common preference will be to be `dynamic_except` if any of the suboperations are that, otherwise `static_except` if any of the suboperations are that, else `no_except`. This is one reason why the enumerator values were chosen as shown, so that for such a function the algorithm for combining the suboperation modes is just `std::max`: `std::max({no_except, static_except, dynamic_except})` does the right thing, and is already `constexpr`.

I expect conditional `throws` to be used less frequently than conditional `noexcept`. Today, conditional `noexcept` has three main uses:

- **(rare, applies also to conditional `throws`) To enable callers to use a different algorithm.** For example, enabling `move_if_noexcept` can allow using a more efficient algorithm while still giving the strong guarantee. The most common use case is for annotating move operations of generic wrappers and containers.
- **(medium, applies also to conditional `throws`) To enable generic intermediate code to preserve the `noexcept`-ness of its implementation.** For example, a `std::transform` call could (but currently is not required to) declare itself `noexcept` if the caller-provided operation is `noexcept`.
- **(very common, does not apply to conditional `throws`) To claw back performance lost to today's dynamic exception handling overheads.** That those overheads are so expensive that we are willing to frequently do tedious programming in the function type system to avoid them (and thereby leak implementation details into the function type) is a strong statement about the unacceptability of today's dynamic exception overheads. In this proposal, new code that otherwise would resort to conditional `noexcept` to avoid the dynamic exception overheads would instead throw statically typed values.

Because conditional `throws` is only for the first two motivations, which are rarer and primarily motivated by generic code that (a) uses move functions or (b) is adaptive to report errors from callees that might use today's dynamic exceptions without converting them to `std::error`, I expect uses of conditional `throws` to be rarer than uses of conditional `noexcept`.

For a function `f` declared with a static-exception-specification `throws(cond)`, then in addition to the rules in §4.1.2:

- `f` behaves as-if `noexcept(!cond)` when queried by the `noexcept` operator and the `*noexcept*` and `*nothrow*` traits (e.g., `move_if_noexcept`, `is_nothrow_move_constructible`).
- `f` must not be virtual.

**Note** In general, trying to mix the always-static computation `throws(cond)` with the always-dynamic `virtual` appears to be a mismatch. The primary known use cases for `throws(cond)` are generic move operations and generic algorithms, which should not be virtual.

However, if we do encounter real examples where this is needed, we can specify it by replacing the foregoing bullet with the following:

- If `f` is a virtual function, then: Every base function that `f` overrides must be declared either `throws` or `throws(cond)` with the identical condition. Every further-derived override of `f` must be declared either `throws(cond)` with the same condition or `noexcept`.

### 4.1.5 Achieving zero-overhead and determinism

*"A big appeal to Go using error codes is as a rebellion against the overly complex languages in today's landscape. We have lost a lot of what makes C so elegant – that you can usually look at any line of code and guess what machine code it translates into…. You're in a statically typed programming language, and the dynamic nature of exceptions is precisely the reason they suck." — [Duffy 2016]*

Recall from §2.5 that today's dynamic exception handling model violates the zero-overhead and determinism principles because it requires throwing *objects of dynamic types* and using *non-local by-reference* propagation and handling semantics.

The primary benefits of this proposal accrue from avoiding overheads by design, by throwing *values of statically known types* and uses *local value* propagation and handling semantics, which eliminate the inherent overheads listed in §2.5 because the proposed exceptions are just local return values:

- Multiple exceptions in flight can be folded by routine optimizations (see Appendix for strawman).
- Exceptions are always allocated as an ordinary stack value.
- Exceptions share the return channel instead of wasting it, including being returnable in registers.
- Exceptions have a statically known type, so never need RTTI.

This let us achieve the zero-overhead and determinism objectives:

- Zero-overhead: No extra static overhead in the binary (e.g., no mandatory tables). No dynamic allocation. No need for RTTI.
- Determinism: Identical space and time cost as if returning an error code by hand.

Furthermore, because the proposed mechanisms are much simpler than today's, they are also more amenable to optimization, including existing optimizations already commonly in use. For example:

- Because exception objects are always entirely local on the stack, and not required to have unique addresses, they are expected to be easy to fold using existing common optimizations.
- In this model, the count maintained by `uncaught_exceptions` is incremented on `throws` and decremented on `catch` as usual, but compensating unread inc/dec pairs are expected to be easier to elide.

> **Note** Whether `uncaught_exceptions` is or can be zero-overhead, including to not use thread local storage, and if not whether to replace it with some other feature that does efficiently support scope guards with zero overhead (e.g., passing an optional flag to a destructor), is a separable question that does not affect the rest of this proposal.

At call sites (that propagate or handle an error), a potential downside of the if-error-goto-handler implementation model is that it injects branches that can interfere with optimizations. However, because this static exception model is isomorphic to error codes (which the dynamic exception model is not), implementations can also choose whether to implement this exception model as error returns or using table-based handling as a pure optimization (no longer a required overhead). And this was tried out in practice in a large native code system on the Midori project, which used a very similar exception design:

> *"A nice accident of our model* [an the exception model that was isomorphic to error codes] *was that we could have compiled it with either return codes or* [table-based] *exceptions. Thanks to this, we actually did the experiment, to see what the impact was to our system's size and speed. The exception*[-table]*s-based system ended up being roughly 7% smaller and 4% faster on some key benchmarks." — [Duffy 2015]*

Again, this is a pure local optimization. Compilers could provide the option to prefer zero binary size overhead (no tables) or fewer local branches in functions (tables), as with other space/time optimization options.

### 4.1.6 Side by side examples

To illustrate, consider these examples from [P0323R3] and [Douglas 2018]. In each case, the right-hand side is expected to have identical or better space and time cost compared to the left-hand side, and identical space and time cost as returning an error code by value.

> **Note** In some cases, such as divide-by-zero in the first example, normally it's best to use a precondition instead. However, I'm preserving the examples' presented semantics for side-by-side comparison.

Checked integer division, showing a caller that does error propagation and a caller that does error handling. Note that in the bottom row we change the `switch` to testing using `==` which performs semantic comparison correctly if the errors come from different domains.

| **P0323R3 example** | **This paper (proposed)** |
|---|---|
| `expected<int, errc> safe_divide(int i, int j) {`<br>`  if (j == 0)`<br>`  return unexpected(arithmetic_errc::divide_by_zero);`<br>`  if (i == INT_MIN && j == -1)`<br>`  return unexpected(arithmetic_errc::integer_divide_overflows);`<br>`  if (i % j != 0)`<br>`  return unexpected(arithmetic_errc::not_integer_division);`<br>`  else return i / j;`<br>`}` | `int safe_divide(int i, int j) throws {`<br>`  if (j == 0)`<br>`  throw arithmetic_errc::divide_by_zero;`<br>`  if (i == INT_MIN && j == -1)`<br>`  throw arithmetic_errc::integer_divide_overflows;`<br>`  if (i % j != 0)`<br>`  throw arithmetic_errc::not_integer_division;`<br>`  else return i / j;`<br>`}` |
| `expected<double, errc> caller(double i, double j, double k) {`<br>`  auto q = safe_divide(j, k);`<br>`  if (q) return i + *q;` | `double caller(double i, double j, double k) throws {`<br>`  return i + safe_divide(j, k);`<br>`}` |

```
   else return q;
}
```

```
int caller2(int i, int j) noexcept {
   auto e = safe_divide(i, j);
   if (!e) {
      switch (e.error().value()) {
      case arithmetic_errc::divide_by_zero:
         return 0;
      case arithmetic_errc::not_integer_division:
         return i / j; // ignore
      case arithmetic_errc::integer_divide_overflows:
         return INT_MIN;
      // No default: Adding a new enum value causes a compiler
      // warning here, forcing an update of the code.
      }
   }
   return *e;
}
```

```
int caller2(int i, int j) noexcept {
   try {
      return safe_divide(i, j);
   } catch(error e) {
      if (e == arithmetic_errc::divide_by_zero)
         return 0;
      if (e == arithmetic_errc::not_integer_division)
         return i / j; // ignore
      if (e == arithmetic_errc::integer_divide_overflows)
         return INT_MIN;
      // Adding a new enum value "can" cause a compiler
      // warning here, forcing an update of the code (see Note).
   }
}
```

**Notes** Pattern matching would benefit both sides of the last row and remove most of the style delta.

The reason to use `if` and `==` (not `switch`) on the right-hand side is to get semantic comparison. Even so, compilers could warn on missing cases on the right-hand side because all the cases are simple tests against `arithmetic_errc` values, and a simple heuristic can deter whether the code checks for all but one or two of the values in a given domain. In the future, replacing the `if`-cascade with pattern matching would restore the ability for the code to express by construction that it is testing `e`'s values, which would make such diagnostics simpler again as with `switch` today.

| [Douglas 2018] example | This paper (proposed) |
|---|---|
| `outcome::result<int> convert(const std::string& str) noexcept {`<br>`  if (str.empty())`<br>`  return ConversionErrc::EmptyString;`<br>`  if (!std::all_of(str.begin(), str.end(), ::isdigit))`<br>`  return ConversionErrc::IllegalChar;`<br>`  if (str.length() > 9)`<br>`  return ConversionErrc::TooLong;`<br>`  return atoi(str.c_str());`<br>`}` | `int convert(const std::string& str) throws {`<br>`  if (str.empty())`<br>`  throw ConversionErrc::EmptyString;`<br>`  if (!std::all_of(str.begin(), str.end(), ::isdigit))`<br>`  throw ConversionErrc::IllegalChar;`<br>`  if (str.length() > 9)`<br>`  throw ConversionErrc::TooLong;`<br>`  return atoi(str.c_str());`<br>`}` |
| `outcome::result<int> str_multiply(const string& s, int i) {`<br>`  auto result = convert(s);`<br>`  if (result) return result.value() * i;`<br>`  else return result.as_failure();`<br>`}` | `int str_multiply(const string& s, int i) throws {`<br>`  auto result = convert(s);`<br>`  return result * i;`<br>`}` |
| `outcome::result<int> str_multiply2(const string& s, int i) {`<br>`  OUTCOME_TRY (result, convert(s));`<br>`  return result * i;`<br>`}` | `int str_multiply2(const string& s, int i) throws {`<br>`  auto result = convert(s);`<br>`  return result * i;`<br>`}` |

### 4.1.7 What we teach

What we teach:

- **Function authors:** Prefer to write exactly one of unconditional `noexcept` or `throws` on every function. Dynamic exceptions and conditional `noexcept` still work as well as ever but are discouraged in new/upgraded code.

> **Note** This is similar to how since C++11 the updated guidance for virtual functions is now to write exactly one of `virtual`, `override`, or `final` on every virtual function (e.g., C.128 in the *C++ Core Guidelines*). We got the default "wrong," but we can now provide a consistent and clear (and mechanically enforceable) style in new code at the cost of writing one word compared to what we could achieve with a time machine.

- **Function callers:** To catch a static exception, write `catch(error)` (note: by value is now fine). The object can be rethrown, copied, stored, etc.
- **Compatibility:** Dynamic exceptions and conditional `noexcept` still work. You can call a function that throws a dynamic exception from one that throws a static exception (and vice versa); each is translated to the other automatically by default or you can do it explicitly if you prefer.

### 4.1.8 How to migrate, and toolability

To migrate/upgrade, you can adopt `throws` on function declarations incrementally, and the more you use it the more you eliminate overheads:

- To upgrade an existing function that is not `noexcept(true)`, write `throws` on all of its declarations (in place of a conditional noexcept-specifier if there was one).
- If your function is on an ABI boundary and currently throws dynamic exceptions, and you have to keep it ABI-stable, then just leave it as-is; you can still start using static-exception-specifications internally within the modules, and they will be translated across the boundary.
- If you have a boundary where you couldn't use exceptions today because of their cost, and can change the boundary, you can now consider reporting errors using exceptions even on that boundary and still freely call any existing internal functions that still throw the old way.

This proposal is amenable to tool support. To facilitate adoption and migration of a code base:

- Compilers can optionally provide a convenience mode that automatically treats every non-`noexcept(true)` function as though it were declared `throws`.
- "Modernizing" tools can mechanically update every non-`noexcept(true)` function to decorate it with `throws`.

> **Notes** Any transformation to reinterpret or rewrite existing code is an ABI change for that code, so it should be explicitly opted-in to remain under user control.
>
> The standard library can use this proposal in four main stages:
>
> (with no change to the stdlib) Users can use the above transformation to make the standard library available in the new mode.
>
> The standard library could consider specifying that, in freestanding implementations, the standard library is available in the above mode.

Update dual `filesystem` APIs to change from:

```
directory_iterator& operator++();
directory_iterator& increment( std::error_code& ec );
```

to:

```
directory_iterator& operator++() throws;
```

Adopt it in other places, such as new library functions whose efficiency is paramount.

Projects that ban today's dynamic exceptions for the efficiency and determinism reasons summarized in §2.5 can continue doing so as long as those reasons continue to apply, but now would be able to enable these light-weight exceptions:

- In their own code's error handling, they can now adopt `throw`/`try`/`catch` with all its benefits. Today's compilers provide modes to turn off today's dynamic exception handling; they would continue to do that, but provide a mode that enables static-exception-specifications only.
- When using STL and the rest of the standard library, my hope is that this proposals lets them adopt "near-normal" STL with all its benefits even without any change to the standard library specification or implementation. For example, because today the standard library reports error using dynamic exceptions, companies (such as Electronic Arts [N2271] and Windows) have resorted to specifying and maintaining a divergent STL that reports errors in different ways (or fails fast), which changes the interface and design of the library. Instead, with this proposal, the aim is to enable a project to use the compiler mode mentioned above (that automatically treats every non-`noexcept(true)` function as through declared `throws`) and just recompile their existing standard libraries in that mode. The standard library compiled in that mode is not strictly conforming, but it is also nowhere near as divergent as today's "no-exception STLs" because the delta is much smaller, all errors that were reported by throwing exceptions are still reported by throwing exceptions (just using the `error` type for exceptions), and the transformation from the conforming standard library is purely mechanical.

### 4.1.9 Discussion: Throwing extensible values vs. arbitrary types

*"[In CLU] We rejected this approach [propagating dynamic exceptions] because it did not fit our ideas about modular program construction. We wanted to be able to call a procedure knowing just its specification, not its implementation." — [Liskov 1992]*

*"[In Midori,] Exceptions thrown by a function became part of its signature… all 'checking' has to happen statically… those performance problems mentioned in the WG21 paper [N3051] were not problems for us." — [Duffy 2016]*

I believe that exceptions of statically known types thrown by value, instead of arbitrary dynamic types thrown by reference, is a natural evolution and an improvement. It follows modern existing practice in C++ (e.g., `std::error_code`, Expected [P0323R3], Outcome [Douglas 2018]) and other languages (e.g., Midori [Duffy 2016]). And, just as when we added move semantics as C++11's marquee feature, it doubles down on C++'s core strength of efficient value semantics.

We've already been learning that it's problematic to propagate arbitrarily typed exceptions, through experience over the past 25 years not only in C++ but also in other mainstream languages. Consider the following overlapping reasons, in no particular order.

**`error_code` values are more composable than types, easier to translate (when propagating) and compare (when handling).** `std::error_code` (especially evolved as described in [P0824R1]) is able to represent and compare equivalent codes from different domains, including all errors reported from the C++ standard library, POSIX, and Windows (with some support for translation between them), plus non-lossy automatic propagation.

**Propagating arbitrarily typed exceptions breaks encapsulation by leaking implementation detail types from lower levels.** ("Huh? What's a `bsafe::invalid_key`? All I did was call `PrintText`!") As a direct result…

**The most valuable `catch`es routinely ignore the type information they do have: Distant `catch`es are frequently untyped in practice.** It has been observed that the value of exception handling increases with the distance between the `throw` and `catch` (Sergey Zubkov and Bjarne Stroustrup, private communication). But because of the previous bullet, the leaked lower-level types are not useful in practice in higher-level code that does not expect or understand them,[4] and higher-level code that writes `catch` often resorts to writing `catch(...)` rather than mentioning specific types from lower levels. — Because the most valuable `catch`es (the distant ones) commonly don't understand the exception's details and ignore their types, throwing arbitrary types has less value in practice in exactly the cases where automatically propagated exceptions are the more effective.[5]

**Propagating arbitrarily typed exceptions is not composable, and is often lossy.** It is already true that intermediate code should translate lower-level exception types to higher-level types as the exception moves across a semantic boundary where the error reporting type changes, so as to preserve correct semantic meaning at each level. In practice, however, programmers almost never do that consistently, and this has been observed in every language with typed propagated exceptions that I know of, including Java, C#, and Objective-C (see [Squires 2017] for entertaining discussion). This proposal actively supports and automates this existing best practice by embracing throwing values that can be propagated with less loss of specific information.

**A `catch(type)` cascade is type-switching.** We have learned to avoid switching on types, and now actively discourage that in our coding guidelines, yet a `catch(type)` cascade is often a form of type switch with all the problems of type switching.[6] — In some cases, especially at lower levels, the code may intend to handle all errors. In that case, when the set of catchable `error`s can be statically known, automatic tools can often determine that we tested "all but one or two" of the possible values of a given error subtype, and so likely meant to test them all; and so when we add a new error value, we can get compile-time errors in code that tries to handle all errors but was not updated to handle the new error value (this will get even better when we get pattern matching). When throwing arbitrary types, the set of potential errors is open-ended (absent whole program analysis, which we can't count on), and so it is much harder for tools to automatically detect a failure of intent to check all errors.

**We've already actively been removing attempts to statically enumerate the types of arbitrarily-typed dynamic exceptions, in C++ and other modern languages.** For example, C++ has already removed trying to enumerate

---

[4] That is, "understand" to handle programmatically, more than just present them to a human such as by logging.

[5] Michael Novak comments on how this aligns well with existing practices using dynamic exceptions: "I agree with this. One 'patch' for this problem that I find frequently employed (we do it ourselves in WIL [Windows Internal Libraries]) is simply normalization of the exception type. WIL strongly discourages creation of any custom exception types of your own. It normalizes on exactly one exception type and then provides helpers and macros to facilitate making it easy for everyone to throw that one type. This has worked very well for us as it allows us to bring along a set of relevant debugging information and normalize how errors are handled. I see this proposal overall being very similar (normalize on a single failure 'exception' type) and quite compatible with what we've been doing for a while now."

[6] And not avoidable in general because exception types can be arbitrarily unrelated. The usual advice to avoid a type switch is to replace it with a virtual function, which requires a tightly coupled class hierarchy that can be extended or modified.

the types of dynamically typed exceptions in function signatures by removing `throw(type1,type2,type3)`, and instead made `noexcept` be untyped (just "can fail or can't fail"). Similarly in Java, real-world uses of checked exception specifications quickly devolve to `throws Exception`. Learning from the variety of error handling methods in Objective-C (described in a nutshell in the first 10 minutes of [Squires 2017]), Swift decided to pursue a paper very similar to this paper's model, throwing a type-erased trivially-relocatable `Error` that always represents an error value (i.e., is not default constructible).

**We've already standardized `std::error_code` and are moving to a world that is at least partly `std::error_code`-based.** [P0824R1] makes this clear, pointing out for example that the actively-progressing `std::experimental::expected<T,E>` and Boost.Outcome are `error_code`-based and are expected to more aggressively embrace `error_code` (or an incremental evolution thereof) as their de-facto standard default code type. Our current status-quo path is to standardize these as library-only solutions, but if we continue down that path we will end up with a result that is still inferior because it's missing language support for automatic propagation and to separate the normal and error paths cleanly.

**Existing practice with error codes already tries to emulate the distinction between normal and error handling logic (using macros) to make the "normal" path clean.** For example, the Windows Internal Libraries (WIL) designed its error-macro-handling system to enable error-code based code to emulate exception-based code by hiding all the branches that are about error handling. For example, real code commonly followed this structure:

```
// "Before" WIL error handling macros

HRESULT InitializeFromStorageFile(IStorageItem *pFile) {
    ComPtr<IStorageItem> spFileInternal;
    HRESULT hr = pFile->QueryInterface(IID_PPV_ARGS(&spFileInternal));
    if (SUCCEEDED(hr)) {
        hr = spFileInternal->GetShellItem(IID_PPV_ARGS(&m_spsi));
        if (SUCCEEDED(hr)) {
            hr = spFileInternal->get_CreationFlags(&m_sicf);
            if (SUCCEEDED(hr)) {
                hr = spFileInternal->get_FamilyName(&m_spszPackageFamilyName);
            }
        }
    }
    return hr;
}
```

So they "hid the error handling behind a wall of macros" as follows:

```
// "After," using WIL error handling macros

HRESULT InitializeFromStorageFile(IStorageItem *pFile) {
    ComPtr<IStorageItem> spFileInternal;
    RETURN_IF_FAILED(pFile->QueryInterface(IID_PPV_ARGS(&spFileInternal)));
    RETURN_IF_FAILED(spFileInternal->GetShellItem(IID_PPV_ARGS(&m_spsi)));
    RETURN_IF_FAILED(spFileInternal->get_CreationFlags(&m_sicf));
    RETURN_IF_FAILED(spFileInternal->get_FamilyName(&m_spszPackageFamilyName));
    return S_OK;
}
```

This attempts to emulate (poorly, by hand and using macros) exception handling's ability to distinguish the clean "success" path and propagate failures.

We *want* C++ programs to propagate errors automatically, and we *want* them to write distinct normal and error handling logic. It's time to embrace this in the language, and effectively "`throw error_code`s" so that the language can (a) solve the performance problems with exceptions by throwing zero-overhead and deterministic exceptions, and also (b) make these new facilities easier to use robustly and correctly via language support for automatic propagation and `throw`/`catch` error paths that are distinct from normal control flow. — Then we can write future `filesystem`-like APIs naturally as

```
directory_iterator& operator++() throws;      // operator + 1 function + 1 way to report errors
```

and the world will be a better place for everyone.

## 4.2 Proposed cleanup: Don't report logic errors using exceptions

| | |
|---|---|
| **SG Poll** | The 2018-04-11 SG14 telecon took a poll on pursuing this direction: 12-1-1 (Favor-Neutral-Against). The one Against was because a previous draft proposed removing the existing throwing behavior, which would be a breaking change; that was demoted to "Future" below to address the concern. |
| **LEWG Poll** | The 2018-06 Rapperswil LEWG session unanimously supported this direction: 26-4-0-0-0 (SF-F-N-WA-SA). They noted [P0788R2] has already been approved unanimously in LEWG (2017-11 Albuquerque) and LWG (2018-06 Rapperswil) as the initial step. |

*"[With contracts,] 90-something% of the typical uses of exceptions in .NET and Java became preconditions. All of the `ArgumentNullException`, `ArgumentOutOfRangeException`, and related types and, more importantly, the manual checks and `throw`s were gone." — [Duffy 2016]*

Even without this proposed extension, there is cleanup that we should do anyway in the standard library that would immediately benefit users. Work on the following direction is already in progress via [P0380R1] and [P0788R1] which in part systematically distinguishes between actual preconditions and recoverable errors, and [P0132R0] which aims to deal with `bad_alloc` (including derivatives like `bad_array_new_length`) in the standard library.

As noted in §1.1, preconditions, postconditions, and assertions are for identifying program bugs, they are never recoverable errors; violating them is always corruption, undefined behavior. Therefore they should never be reported via error reporting channels (regardless of whether exceptions, error codes, or another style is used). Instead, once we have contracts, users should be taught to prefer expressing these as contracts (or assertions or similar tools, not error results or exceptions), and we should consider using those also in the standard library.

The standard library should identify these cases, and aim to eventually replace all uses of exceptions and error codes for such bugs. [P0788R1] is already moving in this direction, by moving toward systematically distinguishing preconditions (et al.) from recoverable errors throughout the standard library.

Then, except for exceptions from user-defined types, the vast majority of standard library operations can be either `noexcept` or throw only `bad_alloc` (see also §4.3). [P0132R0] aims to offer suitable alternatives for code that aims to be hardened against heap exhaustion; with these in place we could consider reporting heap exhaustion differently from recoverable errors (but that is not proposed in this paper).

The following types are used (in part) for preconditions:

```
logic_error  domain_error  invalid_argument  length_error  out_of_range  future_error
```

Note there are non-precondition uses, such as `future_error` being used to report normal errors, and `vector<T>::at` to throw an exception on a failed range check. We may want to keep those behaviors.

For each of the above types:

- Add a non-normative note in the standard that we are maintaining the types' names for historical reasons, but that these types should not be thrown to report precondition failures.
- In every place where the standard library is currently specified to throw one of the above types and that upon review is or should be a precondition, add an equivalent [P0788R2] *Expects:* element.
  - **Future** (to avoid a breaking change now): Someday, remove the corresponding Requires/Throws clause (breaking change). If that eliminates the possibility of throwing, then additionally mark the function `noexcept`. ((And if §4.1 is adopted: Otherwise, mark the function `throws`.))

## 4.3 Proposed cleanup: Treat heap exhaustion (OOM) specially

| | |
|---|---|
| **SG Poll** | The 2018-05-02 SG14 telecon took a poll on treating heap exhaustion as something distinct from other errors (i.e., not report it via an exception or error code) "in some way" (i.e., not tied to a particular method). The poll result was: 9-0-3-0-0 (SF-F-N-WA-SA). |

### 4.3.1 The problem

*"… if STL switched to making bugs [logic errors, domain errors] be contracts… and we could make bad_alloc fail fast, we could use the standard STL unmodified throughout Windows."*
*— Pavel Curtis, private communication*

*"Reducing number of potential exception points is of paramount importance IF we want app-level developers to aim for exception safety." — [Ignatchenko 2018a]*

*"I prefer fail fast securely over an untestable set of failure combinations that might result in us slow failing insecurely." — Fergal Burke, private communication*

The following table summarizes the advice in §1.1 in cases ordered from least to most recoverable to resume successful program operation, and highlight a key question (middle row):

| | **What to use** | **Report-to handler** | **Handler species** |
|---|---|---|---|
| **A. Exhaustion** of the abstract machine (e.g., stack overflow) | Terminate | User | Human |
| **B. Programming bug** (e.g., precondition violation) | Contracts, asserts, log checks, … | Programmer | Human |
| **C. Recoverable error** that can be handed programmatically (e.g., host not found) | Report error (error code or exception) | Calling code | Code |

Is heap exhaustion like all other errors (status quo, as today it is specified to be reported via `bad_alloc` and related types like `bad_array_new_length`), or should it be treated specially? The key differences include:

- Unlike case A (e.g., stack overflow), a heap allocation is always explicitly requested by a function in its source code, and so in principle the function or its caller can test and attempt to respond to the failure.
- Unlike case B, a heap allocation is not a programming bug because the programmer cannot in general test in advance whether it will fail before attempting the heap request.
- Unlike case C, OOM is different in degree than all other errors, because by default any code could transitively request memory allocation; because it is so pervasive, OOM-handling code is not testable using normal methods, and when inspected is commonly found to be wrong. Also, reporting OOM is nonportable, because accurately detecting OOM at the point of the failed allocation is not reliably implementable on all platforms (e.g., default Linux, iOS, Android, macOS).

| | |
|---|---|
| **Notes** | Allocation failure does not necessarily mean no memory is available; a smaller request could succeed.<br><br>See §4.3.5 Discussion comments and notes about applying `ulimit` and lazy commit on Linux. |

### 4.3.2 Making a clear distinction: How to decide (A) vs. (C)

Allocation failure is never (B), a programming bug. This paper proposes that we can clearly decide whether a specific allocation failure is (A) vs. (C) by applying two fundamental questions:

**(1) Can the abstract machine still be expected to execute general purpose code (i.e., the allocation failure does not necessarily imply abstract machine exhaustion)?** For example:

- If `new int[N]` fails where `N` is `10'000'000` or is unsanitized input, the answer is yes. Conceptually, asking for a large buffer is an attempt to acquire a general resource, just like asking for a file handle or a network socket. If that resource request fails, that clearly does not imply that the abstract machine is exhausted; the code can try again with a fallback size, abandon the attempted operation, or do something else.
- But if `new int` fails, so that we couldn't even allocate a few bytes, the answer is no. In that case, we cannot next call most C++ functions – in fact, we cannot even perform a normal `throw` because that is required to allocate an exception object.[7]

**(2) Can the failure path (inside the allocation function and its immediate caller) be expected to be OOM-resilient restricted code?** The default answer is no:

- If the code is known to be carefully written to observe strict OOM-resiliency restrictions along allocation failure code paths, the answer is yes. For example, OOM-resilient code must be guaranteed not to perform any new allocations (including by calling another function, making a system call, or using a language feature such as throwing an exception) along all allocation failure control flow paths (including for destructors during unwinding and code transitively called from those destructors).
- But if there's any doubt, the answer is no. General code is not OOM-resilient.

Because reporting an error (as either `nullptr` or `bad_alloc`) performs stack unwinding, it can only be done reliably if we know we can run that code (the rest of this function, the unwinding, and the caller's handler), which means that at least one of (1) and (2) must be true. If both (1) and (2) are false, we cannot report the failure to the caller.

| | | (1) Can the abstract machine still be expected to execute general purpose code (i.e., it is not exhausted)? | |
|---|---|---|---|
| | | **Yes** | **No** |
| (2) Can the failure path (inside the allocation function and its immediate caller) be expected to be OOM-resilient restricted code? | **Yes** | Report failure to caller (no change to status quo) | Report failure to caller (no change to status quo) |
| | **No** | Report failure to caller (no change to status quo) | |

[7] The standard currently requires that in this specific case the implementation must `throw bad_alloc`. Recall that `throw` is required to dynamically allocate the exception object — in this case, one that is at least as big as the `int` we just failed to allocate. The only reason that throwing `bad_alloc` has ever worked in this case is that implementations are implicitly required to allocate `bad_alloc` objects from a *different* reserve storage that is separate from all other dynamic storage; otherwise, if that `throw` tried to perform a normal `new bad_alloc` it would immediately fail too.

The top right case (No/Yes) is safe because an OOM-resilient allocation function *can* report failure to an OOM-resilient caller: Both the OOM-resilient allocation function body and the OOM-resilient caller are careful to not perform anything but restricted operations along the failure paths.

Note that the lower right case (No/No) contains the large majority of default (`new`) allocations, and naturally maps to single-object allocation using the default allocator.

### 4.3.3 Survey of other modern languages

Here is my understanding of what some other modern languages currently do. (Corrections are welcome.)

| Language | OOM considered recoverable (can report), or catastrophic (must terminate)? | OOM reported as an error/exception? |
|---|---|---|
| **Go** [Go OOM] | Terminate (panic) | No |
| **Rust** [Rust OOM] | Terminate (panic)<br>*[Contemplated: Report via opt-in to an OOM-reporting allocator]* [8] | No<br>*[Contemplated: Yes (`Result`) via opt-in to an OOM-reporting allocator]* |
| **C#** [C# OOM] | Terminate (`FailFast`) | Yes "but not really" – users are told that if they do catch `OutOfMemoryException`, the catch handler should explicitly call `Environment.FailFast` |
| **System C# (Midori)** [Duffy 2016] | Terminate by default<br>Report by opt-in | No by default<br>Yes by opt-in to OOM-reporting allocator |
| **Swift** [Swift OOM] | Terminate for fixed-sized allocations<br>Report for variable-sized allocations | No for fixed-size allocations<br>Yes for variable-sized allocations |
| **C** | Report always | Yes (null) |
| **C++** | Report always | Yes (null, `bad_alloc`) |
| **This proposal** | Terminate by default for fixed-size allocations using the default allocator<br>Report for variable-sized allocations and via opt-in to OOM-reporting allocator | No by default for fixed-size allocations using the default allocator<br>Yes (null, `bad_alloc`) for variable-sized allocations or for an OOM-reporting allocator |

The good news is that there appears to be a convergence among modern languages, many of which (including this paper) are for systems languages that have independently arrived at the same basic default/opt-in distinction criteria, or at least have the same default and are actively discussing the same opt-in and distinction.

This paper has independently arrived at the same conclusions as Midori and Swift in particular; what is proposed herein is effectively merging Midori's and Swift's criteria for deciding which allocation failures are recoverable/reportable.

[8] There is active discussion in Rust about adding an `unsafe` allocator that would return a `Result`. See discussion threads like this one, and this working group that is considering such an allocator (the working group appears to be still ongoing). I have also corresponded privately with other Rust experts who are actively working on such allocators in parallel/complementary efforts. However, all such work is aimed at providing an extension, i.e., to add an opt-in OOM-reporting alternative to the status quo; the Rust default, and the only option for safe code, would continue to be termination (panic).

### 4.3.4 Proposal

This section contains a revised proposal that **still needs discussion and refinement**. I do not consider it "done." It is an iteration toward the goal of finding a clean line where we do not attempt to report abstract machine exhaustion using error reporting (null or `bad_alloc`) when programmatic recovery is not reasonably possible, where today functions that could only fail in such a situation cannot be made `noexcept`, and it is desirable and rational to enable them to be able to be made `noexcept`.

Tis paper proposes the policy that:

- Fixed size default allocation should terminate on failure (after calling the `new_handler` if any), because it is in the No/No case: (1) is No because it is a fixed-size (typically small) allocation, and (2) is No because it is the default for general-purpose code.
- Variable size default allocation reports failure (via `nullptr` or an exception, no change to status quo).
- Custom allocation (including placement `new`) chooses its policy. OOM-resilient allocators report failures.

**EWG and LEWG polls request:** Encourage further exploring the above policy direction?

#### 4.3.4.1 Core language: `new` and `new[]`

For the core language `new` and `new[]`, this paper proposes that:

- Non-placement single-object `new T` is a fixed size default allocation, and so should terminate on failure (after calling the `new_handler` if any). (**This is the key proposed change to status quo.**)
- Non-placement array `new` is a variable size default allocation, and so should report failure via `bad_alloc`. (**No change to status quo.**)
- Single-object placement `new(nothrow)` is a custom allocation (so could make either choice), and is de facto already used as an OOM-aware allocator (because code that calls it is assumed to check the return value), and so should continue to report failure via `nullptr`. (**No change to status quo.**)
    - The return value of `new(nothrow)` should be treated as `[[nodiscard]]` if it is not already.

**EWG poll request:** Encourage further exploring the above core language change to single-object `new`?

While this would enable dropping the requirement that today's dynamic exceptions use a different allocator than `operator new`, this paper does not propose that, because if §4.1 is adopted it won't be necessary.

#### 4.3.4.2 Standard library: `std::allocator` and containers

For `std::allocator`: `std::allocator` obtains storage by calling `::operator new`. If the implementation always uses the array form of `new`, then its behavior will be unchanged. The key drawback of today's Allocator requirements and `std::allocator` is that they do not have the `new`/`new[]` distinction; they have only `allocate(N)` which is a variable-sized allocation that conflates `allocate(1)` into the same operation. This paper proposes that we:

- Extend the `Allocator` interface requirements with `allocate_one()`, which allocates a single object. Because of question (1), this is an important distinction to surface in the allocator interface.
- Extend `std::allocator` with `allocate_one()` that calls single-object `new` instead of array `new`, and will therefore terminate on failure.

For containers: Containers themselves can conceptually be viewed as custom allocation arenas. That is, `container.push_back` and `container.insert` are effectively allocation functions on the memory arena `container`. We should review their semantics in light of questions (1) and (2). There are three main direct allocation request cases:

- For all containers, all range `insert`/construction/assignment and `reserve`/`resize` operations are variable-sized allocations. So the answer to question (1) is Yes, failure to allocate *does not* imply heap exhaustion, and this is what would happen under this proposal for containers that use `std::allocator`, they would continue to throw `bad_alloc`. (**No change to status quo.**)
- For contiguous containers, an individual single-element `insert`/`push`/`emplace` operation is apparently a single-object allocation, but actually performs a variable-sized allocation. So the answer and reasoning is the same as in the previous bullet, they would continue to throw `bad_alloc`. (**No change to status quo.**)
- For node-based containers, an individual (non-range, single-element) `insert` operation is a single-object allocation for a new node. By default that would mean the answer to question (1) is No, failure to allocate a single new node *does* imply heap exhaustion. However, the `insert` interface already returns a boolean that returns `false` on failure, so in practice it is designed to be an OOM-aware allocator and code should be checking that. — *Hypothesis: Existing code is generally OOM-safe on node-based single-element insertion failure. If yes, let node-based individual insert operations be unchanged, and continue to call variable-sized allocation. (**No change no status quo design or implementation.**)*
    - The boolean part of the return value of `insert` should be treated as `[[nodiscard]]` if it is not already.

To validate/invalidate the last bullet's hypothesis, the proposer could do this experiment:

- Instrument node-based containers' single-element insert operations to inject allocation failures, and see whether existing code is typically robust or typically fails.

If the result is that existing code typically crashes, that would a reason to reconsider whether node-based container single-element insert should call `allocate_one` instead.

My hope is that LEWG can agree that this or other experiments are useful, and what results would be likely to influence the group's opinion. That would help justify doing the work, instead of doing it and then coming back only to discover that we tested something that wasn't useful to inform the group.

**LEWG poll request:** Encourage further exploring the above direction? Or encourage exploring more aggressive options for terminating on allocation failure?

Finally, fix `bad_array_new_length` (see Discussion): Cases that are specified to throw `bad_array_new_length` should first invoke the `new_handler`, and then throw the exception only if the handler returns. This permits handlers that, for example, terminate (suggested by Mathias Stearn) or log information (which would be useful to document an exploit that was thwarted, in which case it would likely also terminate), and likely other uses. (Note that the handler cannot throw a different type; handlers must throw `bad_alloc` or a derived type.)

**EWG and LEWG poll request:** Cases that are specified to throw `bad_array_new_length` should first invoke the `new_handler`, and then throw the exception only if the handler returns?

### 4.3.5 Additional notes and discussion

> *"C++ aspires to making out-of-memory a recoverable condition, and so allocation can throw. Therefore, it is essentially compulsory for the language to assume that constructors might*

*throw. Since constructors are called pervasively and implicitly, it makes sense for the default rule to be that all functions can throw… This all adds up to a significant code-size penalty for exceptions, even in projects which don't directly use them and which have no interest in recovering from out-of-memory conditions. For this reason, many C++ projects explicitly disable exceptions and rely on other error propagation mechanisms, on which there is no widespread consensus." — [McCall 2016]*

**Some code is correct today for heap exhaustion even without coding for it specially and should continue to work.** For example, in a large call tree when the millionth `vector::push_back` fails, we can generally unwind and recover just fine today with just a `catch(...)` at the top. This appears to be particularly in cases that are larger requests, like `vector::push_back`. This proposal does not propose any change to reporting these via `bad_alloc`.

**Testing OOM handling is much more difficult because allocation is pervasive, and lots of code that thinks it handles OOM is wrong.** It is extraordinarily more difficult to test every failure point for heap exhaustion than to test every failure point for most other kinds of errors. Real-world code can be assumed to be not properly testing those failure points, unless it is tested pervasively using technologies like fuzzer-based fault injection combined with long-haul stress testing. Most code that thinks it handles heap exhaustion is probably incorrect; there is some truth to the mantra "if you haven't tested it, it doesn't work." Across many years and organizations, hand inspection of OOM-recovery code (code that exists specifically and only to recover from heap exhaustion) commonly finds it to have never been correct. In released commercial products that are known to be robust and stable in broad use, simulating OOM by making a single random allocation request fail commonly (>90%) crashes the application (unpublished research).

**`bad_alloc` hypothesis test framework.** The previous paragraph mentions a hypothesis we can test. Restating:

*Hypothesis: Existing C++ code that is believed to be correct for OOM is often not actually correct.*

To gather more information about this, in April 2019 Marshall Clow and I created the [babb] library ("`bad_alloc` Behaving Badly"), which randomly injects allocation failures into a C++ application via both `nullptr` and `bad_alloc`. Here are my main initial results and learnings so far.

The first thing I learned is that I had trouble finding candidates to test, namely code bases that were thought to be OOM-resilient on at least some paths. Many teams, including the Visual C++ compiler team and most Office applications, responded that they make no attempt to be OOM-resilient and just terminate (usually after some last-ditch handling appropriate for a `new_handler`), so there was nothing interesting for the failure injector to test for the purpose of this paper (it would only test the correctness of last-ditch handling code, which is not interesting for this paper).

We did successfully run experiments with two code bases:

- **Visual C++ standard library.** C++ standard library implementations are the world's most widely-used C++ libraries, and are generally the most expertly- and carefully-written for exception safety. The library team responded that the VC++ library is carefully written for general exception safety, but not necessarily for OOM-resiliency. I decided to try it out, and the very first test experiment I tried was:

  ```
  try { vector<string> v; for(int i=0; i<N; ++i) v.push_back(alphabet); }
  catch(...) { cout << "caught something"; }
  ```

  When I injected allocation failures with moderate values of `N`, this program consistently aborted (without unwinding or reaching the `catch` block) in VC++'s STL in default debug build, which includes iterator

debugging. This is because `vector` and `string` move operations are now `noexcept`, but the existing implementation still allocates proxies in iterator debug mode.[9] Although this crash is due to a conformance bug, it still yielded useful data: Upon further investigation, the bug has been in the product since at least 2015, and we found only six bug reports we received about it during 2015-2019, even though we estimate that ~90% of customers do not override the default and leave STL iterator debugging mode enabled in their debug builds. — The fact that a common STL operation like `vector<string>::push_back` crashes consistently in our shipping product in the presence of `bad_alloc` in default debug builds, yet this appears to be reported as a bug only twice per year, is initial data leaning against the common existence of `bad_alloc`-hardened customer code:

  - Very few customers who use our STL appear to be encountering `bad_alloc` at all during testing, otherwise common operations would crash and reports would not be so rare.
  - Very few customers who use our STL appear to be writing `bad_alloc`-safe code, otherwise it would have crashed when they OOM-tested it. If they tried to write `bad_alloc`-safe code but did not OOM-test it, it should be presumed incorrect and not actually `bad_alloc`-safe. (Note: An alternate explanation would be that they encountered the problem but routinely reacted by just not using our library in OOM-safe code paths, without complaining about the limitation, which is possible but unlikely.)

- **PowerPoint.** Although most Office code and applications are designed to terminate on OOM, PowerPoint was considered to be OOM-resilient for certain operations including File > Open. The initial pre-experiment comment from the PowerPoint team was: *"if a scenario is exception safe, like opening a document or running a command, we'll fail the command gracefully and continue regardless of exception. So, OOM would actually recover gracefully there, implicitly."* — Then the team did the experiment of trying the [babb] library with PowerPoint to test File > Open. The team reported: *"The file simply failed to open but the application was stable. Most of the time. The problem lies with `noexcept`… [example function] is `noexcept` and yet [directly] calls a throwing `new`… This is obviously going to crash. There are a quite few other places like this."* — The result of the experiment was that the code paths in PowerPoint that were thought to be OOM-safe *"are **not** actually OOM-safe, at least because of the `noexcept`s."* (We did not do further investigation to see if there were any other failure modes besides tripping over `noexcept`, such as termination due to multiple exceptions when babb injected multiple consecutive failed allocation attempts.)

**`noexcept` is one common cause:** Note that in the cases of observed-but-unexpected failure (VC++ STL `vector` and `string` move operations, and PowerPoint File > Open), the cause was the addition of `noexcept` to functions that nevertheless needed to perform allocation under the covers, either directly or transitively. This is a pattern I expected to see as we adopt `noexcept`, because the long-term pressure is (rightly) to make more functions `noexcept`, with is fundamentally in tension with the fact that allocation is pervasive.

**Programs don't run forever; the highest reliability comes from embracing termination, not applying increasingly heroic measures to prevent it.** I sometimes hear that there are programs for which termination is unacceptable; I strongly agree with that premise, except suggest `s/termination/unavailability/`. The reality is that applications can be terminated all the time – by power outage, by hardware crash, by slow memory leaks, by gradual memory fragmentation, by the OS for arbitrary reasons including <*drum roll*> the Linux OOM killer

[9] This bug will be fixed in the next version that takes an ABI break. Both libstdc++ and libc++ handle this initial test fine even with `bad_alloc` injection; they correctly unwind and deliver the `bad_alloc` safely to the `catch` block. Still, even though I expected that regular code was OOM-unsafe and would be likely to fail, I was still surprised that that my very first test using a common `std::` function crashed one major implementation.

itself because if some *other* process you have no control over hits OOM it may be *your* process that gets killed at an arbitrary point in order to recover memory (i.e., even if your code really is perfectly OOM-resilient, it can be terminated arbitrarily by OOM… in someone else's process). — I agree that C++ must support writing high-availability programs. However, my understanding is that the most robust high-availability services do not achieve that availability by trying to eliminate sources of failure (which works to a point, after which increasingly heroic measures result in diminishing returns), but rather *by embracing failure/termination as normal* and engineering for routine restart of loosely coupled components/processes. This seems to be the consistent experience in fields ranging from cloud datacenters (success is lots of machines that routinely fail and failover as needed, not trying to make 'failure-proof' monster machines), to interplanetary spacecraft, to component-based software design (e.g., see [Duffy 2016], search for "processes" to see the quick highlights, 11 hits).

**Portable code cannot rely on OOM reporting because status quo ISO C++ nullptr/`bad_alloc` allocation failure reporting is not reliably implementable on all platforms.** As I wrote in [Sutter 2001]: (a) Reporting heap allocation failure is useless on systems that don't commit memory until the memory is accessed, which is the default on Linux with `vm/overcommit_memory` mode `0` (see [LKA 2018], [Landley] Linux Memory FAQ, and [Gorman 2007] Chapter 13 and page 686's annotated source for `vm_emough_memory()`), because on such systems `new` never actually fails (it might as well be `noexcept`) and exhaustion actually manifests later as a hard fault on an ordinary operation that attempts to access the apparently-successfully-allocated memory (if the OOM killer selects this process); this is nonconforming, but it's actual real-world behavior on such systems.[10] Now that macOS and iOS are 64-bit, it is no longer reasonably possible to get `bad_alloc` on those systems. (b) Allocation failure might never manifest in practice on virtual memory systems, because the actual symptom of nearing heap exhaustion is often increased thrashing that slows the system more and more so that the program never reaches actual heap exhaustion at all. — If we re-specified allocation failures to result in `terminate` or be undefined behavior, that would appear to be an impossibly large breaking change on paper, yet such programs and environments would never notice the difference.

**Recovery requires special care.** Recovery from heap exhaustion is strictly more difficult than from other kinds of errors, because code that is correctly resilient to heap exhaustion can only use a restricted set of functions and language features. In particular, recovery code typically cannot ask for more memory, and so must avoid calling functions that could allocate memory, including functions that could fail by throwing another of today's dynamic exceptions which would require increased allocation. Real-world systems that do handle heap exhaustion correctly are typically written to explicitly use `new(nothrow)` to opt into testing the result, which is not changed by

[10] This is enabled by default in Linux (e.g., Ubuntu, Red Hat Enterprise Linux, macOS, iOS, and Android), and these overcommit semantics are used in real-world production systems, such as at Google. Android has explored disabling it. Even so, in these systems there is no strict commit charge accounting as there is in Windows.

Even with `ulimit`, users report that bad_alloc is not reported reliably. For example, see [StackOverflow 2010], "Why does my program occasionally segfault when out of memory rather than throwing bad_alloc?" It says in part: "I use bash's `ulimit` command to limit the amount of virtual memory the process can use… Certain algorithm/test instance combinations hit the memory limit I have defined. Most of the time, the program throws an std::bad_alloc exception, which is printed by the default handler, at which point the program terminates. Occasionally, rather than this happening, the program simply segfaults."

It is possible for a program on an overcommit system to observe `bad_alloc` by exhausting its address space, such as a 32-bit process on a 64-bit system or on a 32-bit system that does give access to the whole address space, if the address space exhaustion or fragmentation is reported as allocation failure.

this proposal; and in some cases they already use functions such as those proposed by Ville Voutilainen in [P0132R0] that report allocation failures distinctly (e.g., `try_reserve` instead of just `reserve`).

**Reporting preference is often different.** For example, in [Koenig 1996] Andrew Koenig argued why halting programs on allocation failure is the best option in most situations, citing examples like the one in the previous bullet above.[11] Treating allocation failure distinctly is also one of the two major motivations for §4.4 (see second bullet for a real-world example) which is an approach that could incur small but nonzero overheads on all code and might be more readily rejected if we treated heap exhaustion distinctly from other errors.

**We have some precedent in the C++ standard for treating heap exhaustion differently from other errors.** See [syserr.syserr.overview]: "[*Note*: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type `bad_alloc` (21.6.3.1) rather than `system_error`. — *end note*."

**Contracts and heap exhaustion combined outnumber all other failure conditions by ~10:1.** For example, see [Duffy 2015]. Therefore, not reporting them as errors can greatly reduce the number of functions that can fail (emit exceptions or error codes at all). Corollaries:

- **~90% of functions could be `noexcept`, including in `std::`.** Once we ignore exceptions that should be preconditions, `bad_alloc` is the only exception that many standard library operations throw. The combination of changing standard library preconditions to some method other than throwing (e.g., contracts, assertions) and treating heap exhaustion separately means that a large number of standard library and user functions could be made `noexcept`. In existing practice today, we have "exception-free" STL dialects that fail fast on `bad_alloc` as the basis for claiming they do not throw exceptions (unless thrown by user-defined functions types passed to that STL implementation). Based on experience in languages like Go [Alper 2019] and Rust, when OOM and preconditions are not reporting using exceptions, slightly over 90% of all functions do not report errors (are no-fail).
- **Enabling broad `noexcept` would improve efficiency and correctness (and remove most try-expressions, see §4.5.1).** Being able to mark many standard library and user functions as `noexcept` has two major benefits: (a) Better code generation, because the compiler does not have to generate any error handling data or logic, whether the heavier-weight overhead of today's dynamic exceptions or the lightweight if-error-goto-handler of this proposal. (b) More robust and testable user code, because instead of examples like GotW #20 [Sutter 1997] where today a 4-line function has 3 normal execution paths and 20 invisible exceptional execution paths, if we reduce the number of functions that can throw by 90% we directly remove 90% of the invisible possible execution paths in all calling code, which is an important correctness improvement: All that calling code is more robust and understandable,[12] and also more testable because its authors have fewer execution paths to cover. (Using `noexcept` more pervasively today also opens the door wider to entertaining a future C++ where `noexcept` is the default, which would enable broad improvements to optimization and code reliability.[13])

[11] Another example is that there may not be enough memory to even report a `bad_alloc`, such as if on the [Itanium ABI] the fallback buffer gets exhausted, or if on Windows there is insufficient pinnable stack space to store the exception (see Appendix). However, this paper's static exceptions can resolve this particular issue on all platforms.

[12] This also makes it more feasible to adopt §4.5.1 to make those exceptional paths visible, because it will remove 90% of the places to write a "this expression can fail" `try`-annotation.

[13] Of course, changing the default would be a broad language breaking change and would need to be done on some backward-compatibility boundary. The point is just that it's potentially desirable but probably not worth doing unless we could get to a place where most functions are noexcept, and exceptions thrown for preconditions and `bad_alloc` are the two major things that stand in the way of that today for the standard library.

- **Treating heap exhaustion the same as all other errors violates the zero-overhead principle by imposing overheads on all C++ programs.** As noted above, it's common to have platforms and programs where recovering from heap exhaustion is either not possible or not needed, but programs and programmers in that camp are paying for the current specification in performance across their programs. For example, using `noexcept` less often on functions that can only report `bad_alloc` incurs overheads on all programs as described in the previous bullet. As a specific example, resizing `vector<optional<T>>` is potentially slow only because of possible allocation failures. Titus Winters reports that one of the subtle things [Abseil] does that they are most happy with is to provide an explicit build flag for "does my default allocator throw" and to rely on "move constructors are assumed not to throw for anything other than allocation;" the result of that combination is that many move constructors can be made unambiguously `noexcept` and they measure better performance as a direct result (example: abseil::optional).
- **Trying to report and handle heap exhaustion (OOM) *by throwing a heap-allocated exception* is inherently problematic.** To make it work, libgcc's C++ support allocates 32KB of emergency heap to always be able to allocate a `bad_alloc` object in heap exhaustion scenarios. On Azure Sphere (cloud IoT), which uses libgcc, device applications can have 64KB to 256KB of available RAM; using unmodified libgcc would mean that customers' applications would be forced to reduce their available RAM by 12% to 50% just to be able to try to handle heap exhaustion gracefully, even in applications that never actually try to catch or handle that failure. Therefore, Sphere had to remove that emergency reserve from their version of libgcc, which means reporting heap exhaustion is in practice not supported at all. The Sphere team reports that with the model proposed in the following subsection they would be able to support heap exhaustion handling efficiently in their environment.

**`bad_array_new_length` derives from `bad_alloc` but does not invoke the `new_handler`.** This paper's position is that it's fine for `bad_array_new_length` to be viewed as a refinement of `bad_alloc`, and not as a precondition. But not invoking the `new_handler`, if present, is inconsistent and there are several related issues:

- Mathias Gehre noted on lib-ext@ that this *"means that applications that try to convert allocation failure into termination* [by installing a terminating `new_handler`] *can still get `bad_alloc` exceptions thrown through unsuspecting code."*
- Stephan T. Lavavej noted on lib-ext@ that [LWG 3038]'s current proposed resolution for `polymorphic_allocator::allocate` nonsense-sized requests is to throw `length_error`, but he is reporting this as incorrect and instead proposes *"throwing `bad_array_new_length`, and [changing] many other parts of the STL to throw `bad_array_new_length` for arithmetic overflow (upgraded from `bad_alloc`)."* This paper proposes that throwing `bad_array_new_length` makes sense for LWG 3038 specifically which involves an allocator. However, this paper proposes any arithmetic overflow that is effectively a precondition should not be an exception, see §4.2.

**Essentially nobody seems to use `bad_array_new_length` as anything but a synonym for `bad_alloc`.** That is consistent with this paper, which proposes that the default allocator only report `bad_alloc` on variable-length (array) allocations including the case of sizes that may be unsanitized which is a primary case for this exception.

Doing a code search using codesearch.isocpp.org (with thanks to Andrew Tomazos), a codesearch for bad_array_new_length of ~2.5 million source files gets 33 hits:

- 21 hits in `bad_array_new_length` implementations themselves
- 10 hits in "test" code of those implementations
- 2 hits in the libstdc++ function to actually throw one (__cxa_throw_bad_array_new_length)

And that's all. In that code corpus, there are zero (0) uses in real world code. Any code that catches `bad_array_new_length` must be doing it via the `bad_alloc` base class.

Then, as a next step, I looked for implementations and uses of the function that actually throws one…

**… All implementations of `__cxa_bad_array_new_length` in the codesearch.isocpp.org corpus already terminate conditionally or unconditionally.** Interestingly, when I did a codesearch for that function (`__cxa_throw_bad_array_new_length`), I got 9 hits:

- 4 hits are just declarations of that function
- 2 hits are implementations of that function to *conditionally throw or terminate()* depending on a mode
- 2 hits are implementations of that function to make it *unconditionally MOZ_CRASH()*
- 1 hit in "test" code

So, all of the implementations in that corpus already terminate either conditionally or unconditionally.

Finally, a GitHub search for bad_array_new_length finds hits in only 13 repositories (as of June 2019).

**Existing practice.** In early discussions, I have found that this proposal fits actual existing practice much more pervasively than I expected, and in fact has been repeatedly reinvented:

- Code bases that use "exception-free STLs" already often terminate on allocation failure. (Example: Google's production code uses compiler configuration that behaves as if every `new` were annotated `noexcept`, including the ones in `std::allocator`, as part of the strategy to enable using STL in an `-fno-exceptions` environment.)
- It aligns with existing practice on systems like Linux where virtual memory overcommit means that our current C++ standard heap exhaustion design is fundamentally unimplementable and already ignored in practice by default, where `bad_alloc` can never happen and `new` is already de facto `noexcept`.

### 4.3.6 What real-world code would do to adapt to the proposed change

Here are examples of what real-world code would do to adapt to the new model:

- **Program that never encounters heap exhaustion.** No action needed, won't notice the change.
- **Program that doesn't correctly handle `bad_alloc`.** No action needed, won't notice the change.
- **Code that performs one big allocation, e.g., `image.load("very_large.jpg")` or `allocate(large_bufsize)`.** No action needed, no change in behavior because this will call either default `operator new[]` or a custom allocator, both of which do not change behavior.
- **Microsoft Excel, File > Open `huge_spreadsheet.xlsx`: Big operation that causes lots of little-but-related allocations.** Today, 32-bit Excel can encounter heap exhaustion when enterprise users try to open huge spreadsheets. The status quo way it is handled is to display a dialog to the user, and for support to recommend using the 64-bit version of Excel. In this model, only a minimal change is needed to preserve the same behavior: Install a `new_handler` that invokes the existing failure dialog box display code. If the program wishes to continue if the whole large operation fails, it can make the allocations performed as part of the large operation use nonthrowing allocations (`new(nothrow)` and `try_` functions).
- **Microsoft Office shared components.** Many shared components in Microsoft Office are using nonthrowing user types with STL in `noexcept` contexts, with the knowledge that the only exceptions are heap exhaustion and contract violations. These components would be unchanged except their code would be slightly more efficient as the STL calls become `noexcept`.

- **Azure Sphere.** Sphere would be able to keep the libgcc emergency heap reserve turned off and still enable heap exhaustion reporting and handling in customer applications that wanted to handle it, with zero overhead on applications that do not.

| | |
|---|---|
| **SG Poll** | The 2018-05-02 SG14 telecon took a poll whether to pursue this approach to treating heap exhaustion distinctly from other errors. The poll result was: 2-6-3-2-0 (SF-F-N-WA-SA). |
| **LEWG Poll** | The 2018-06 Rapperswil LEWG session supported adding `try_` versions of functions that could allocate memory: 16-13-1-1-0 (SF-F-N-WA-SA), and unanimously supported the direction of changing `bad_alloc` to terminate by default: 20-11-0-0-0 (SF-F-N-WA-SA). |

In summary, this paper proposes the following resolution:

| | **What to use** | **Report-to handler** | **Handler species** |
|---|---|---|---|
| **A. Exhaustion** of the abstract machine (e.g., stack overflow, single-object default allocation) | Terminate | User | Human |
| **B. Programming bug** (e.g., precondition violation) | Contracts, asserts, log checks, … | Programmer | Human |
| **C. Recoverable error** that can be handed programmatically (e.g., host not found, array default allocation and OOM-aware custom allocator) | Report error (error code or exception) | Calling code | Code |

## 4.4 Optional extensibility hook: `set_error_propagation()`

**Note** This subsection is included because some users have requested the following functionality, especially if we do not adopt §4.3 to treat heap exhaustion separately in §4.3. If this is needed (e.g., because we didn't adopt §4.3) and if it cannot be implemented in a totally zero-overhead way, then it could be a conditionally-supported feature.

We can additionally allow the program to register a global function to allow customizing what happens when exiting each function body with an error, intended to be used only by the owner of the whole program. This is useful for several purposes:

- **To integrate with an existing system's error handling or logging policy.** Niall Douglas reports regarding a similar hook in his implementation: *"I've already had some big multinationals who are thrilled with this feature because they can encode the bespoke failure handling system they are already using into a custom policy, specifically custom logging and diagnostics capture."*
- **To enable fail-fast throughout a system, even when invoking the standard library or third-party code.** For example, the layout engine for Microsoft's [Edge] web browser (EdgeHTML) is designed to `terminate` if memory is ever exhausted. Today, EdgeHTML depends on a nonstandard fork of the standard library that does not throw exceptions and that terminates on `bad_alloc`. This is undesirable not only because it means the Visual C++ team receives requests to support an incompatible nonstandard variant of the standard library at least in-house, but because there is pressure to deliver the same products internally and externally and so there is pressure to document and ship this nonstandard mode which is undesirable for the community. Instead, with this hook EdgeHTML can accomplish its goal by using the standard STL, built in the mode described in §□ where all non-`noexcept(true)` functions are treated as `throws`, and installing a callback that calls `std::terminate()` on `ENOMEM` specifically and does nothing for all other errors. (But see also §4.3.)

Following the model of terminate handlers, we provide the ability register a callback, which is expected to invoke the previously installed callback:

```
using on_error_propagation = void (*)(std::error) noexcept;

atomic<on_error_propagation> __on_error_propagation = nullptr; // global variable

on_error_propagation set_on_error_propagation( on_error_propagation f ) {
    return __on_error_propagation.exchange(f);                 // atomically set new hook
}
```

When exiting a `throws` function, the function epilog performs a jump to this common code, where `error e`, the `unwinding` flag, and the `return_address` are assumed to be already in registers:

```
// pseudocode

if (unwinding)
    [[unlikely]] if (auto x = __on_error_propagation.load(memory_order::relaxed)) x(e);
jmp return_address
```

**Notes** Overhead is expected to be minimal, but (importantly) non-zero. The expected code size cost is one unconditional `jmp` instruction in the epilog of a function declared `throws`: The return address will already be in a register, so just jump to the above common hook code (which will be hot in L1$) which when finished jumps to the return address. The expected L1I$ overhead will be a constant 8-

16 bytes for the hook code above + 1 instruction per `throws` function. The expected L1D$ overhead will be one pointer (the hook target).

For example, here is how to install a callback that will cause all heap exhaustion failures to fail-fast:

```
g_next_handler = set_on_error_propagation( [](error e){
        if (e == std::errc::ENOMEM) terminate();
        if (g_next_handler) g_next_handler(e);
        // else return == no-op, continue propagating
    } );
```

## 4.5 Proposed extension: `try` and `catch` (addresses §3.1 group B)

| | |
|---|---|
| **SG Poll** | The 2018-04-11 SG14 telecon took a poll on pursuing these particular sugars: 5-1-8 (Favor-Neutral-Against). Some of the Against votes would like different sugars; I requested suggestions by email but have not received any yet as of this writing. |

*"The `try` keyword is completely redundant and so are the { } brackets except where multiple statements are actually used in a try-block or a handler." — [Stroustrup 1994]*

*"Failure to standardise this [operator try] means people may abuse `co_await` to achieve the same thing" — [P0779R0]*

| | |
|---|---|
| **Notes** | Several reviewers felt strongly that this should be in the core proposal. For now and unless SG14 or EWG directs otherwise, I'm keeping it distinct; nothing in the core proposal depends on this. |

### 4.5.1 `try` expressions and statements

Today, exceptional error handling flow is invisible by default; between the `throw` and the `catch`, all propagation is via whitespace. This makes it more difficult to reason about exceptional control flow and to write exception-safe code. One of the biggest strengths of `expected<T,E>` and Boost.Outcome is that they make intermediate error-handling control flow *visible* rather than invisible; but at the same time, they make it *manual* rather than automated. This section aims to make it both *visible* and *automated*, to help reduce today's programmer mistakes and so that I don't have to write articles like GotW #20 [Sutter 1997]. — See also related proposal [P0779R0].

This section proposes a **try-expression** and **try-statement**, where the keyword `try` can precede any full-expression or statement of which some subexpression could throw. [14] When applied to an expression, it applies to the maximal expression that follows (in an expression, `try` has the same precedence as `throw`). When applied to a statement, it is primarily useful for variable declarations that invoke fallible constructors.

For example:

```
string f() throws {
    if (flip_a_coin()) throw arithmetic_error::something;
    return try "xyzzy"s + "plover";                 // can grep for exception paths

    try string s("xyzzy");                          // equivalent to above, just showing
    try return s + "plover";                        //   the statement form as well
}

string g() throws { return try f() + "plugh"; }     // can grep for exception paths
```

| | |
|---|---|
| **Notes** | This becomes even more attractive if §4.3 is adopted to dramatically reduce the number of potentially-throwing expressions and therefore the number of places one would write `try`.<br><br>[D0973R0] and other papers have proposed using `co_await` here instead of `try`, and with explicit support for `expected<T,E>` as an endorsed return type to be widely used. |

[14] Swift, and the Microsoft-internal Midori project [Duffy 2016], also created a similar `try`. Go [Cox 2018] is currently considering it. Unlike Swift, my interest is currently in plain `try` on an expression or statement, and not in Swift's additional `try?` (monadic "auto-`optional`") and `try!` ("auto-die") variations.

> In general I agree with the overall direction, namely: Just as the `expected<T,E>` proposal motivates this paper's proposal to bring such a concept into the language as `throws`, I think that [D0973R0]'s suggestion to have an expression-level way to test the error result as `co_await` motivates this section's proposal to bring such a feature into the language as `try` on expressions and statements. I think that is better than trying to shoehorn the feature into `co_await` which is not about error handling, and would leave error-handling code interspersed with normal code — in fact, trying to reuse `co_await` for this really is another example of trying to use normal control flow constructs for error handling, whereas error-handling paths should be distinct from normal control flow. See [P0779R0] section 1.2 for additional reasons why `co_await` is not a good idea here.
>
> Jason McKesson notes that having try-expressions "opens up the possibility of employing `operator try` over ValueOrError types, thus allowing them to be mostly isomorphic with static exceptions. So if you have template code, where the user-provided code could be using static exceptions or `outcome` (or similar types), you can do `try` *expression* on that code. If it used `outcome`, then that types `T operator try() throws;` function will return the value or throw the static exception. If *expression* instead directly used static exceptions, then it just passes them through, and we can see the exception flow pathway. So if people start putting `try` *expression* into their code to mark exception pathways, then we have another reason to allow people to "unpack" ValueOrError types. The two features complement each other." — While I would personally prefer not proliferating such error handling types (see §2), if it turns out they are useful then this would be another point of convergence to allow their use without diverging the programming model, which is the essential thing to converge.

#### 4.5.1.1 try-expressions and function arguments

When `try` appears on an expression used to initialize an argument, it applies also to implicit construction of the argument. This is a great aid to code readability in existing problem cases we know are hard to teach. Consider this canonical example:

```
// existing C++ guru meditation question: is this "noexcept" a lie?
auto f(string s) noexcept { return s.length(); }
```

On the one hand, f's `noexcept` is perfectly legitimate because nothing in `f`'s definition can throw, and the language enforces that no exception can be emitted by this function (else we `terminate`). Nevertheless, arbitrary *call sites* can indeed observe exceptions being thrown (without termination):

```
// both of these call the noexcept function, but the call can throw
f({"xyzzy", 5});
f(mystring);
```

Even though these callers knows that `f` is `noexcept`, and that the language enforces no exception can be thrown, these calls nevertheless can throw because argument evaluation happens before the call. This is nonobvious from the source code, and today this is difficult to teach and remember, and is sometimes a source of ridicule ("look, here's another C++ weird example… `noexcept` function calls can throw anytime, C++ `noexcept` is { useless | broken | designed by committee | <alternative pejorative> }"). Today, if we want to make the code clear, our only option is to write the type construction explicitly:

```
// both of these are clearer that there's a step that can throw, but repeat the type
f(string{"xyzzy", 5});
```

```
f(string{mystring});
```

Using a try-expression, and using the rule that `try` can appear on the initializer of an argument and cover the implicit construction without repeating the type, we can write just the original code plus `try`:

```
// (proposed)
f(try {“xyzzy”, 5});
f(try mystring);
```

With the minimum possible ceremony, this takes the code example that today is opaque and difficult to teach, and makes it totally clear: Yes, `f` is a noexcept function and won’t throw an exception, but yes, you are also performing another call that could throw as you are calling `f`, so now if you get the exception there is no surprise about where it came from.

#### 4.5.1.2 `try`-expressions and data member construction

Consider a type with a fallible constructor that is declared as a data member of class with an in-class initializer, for example:

```
class C {
    string s = “xyzzy”;
    // ...
};
```

The initialization that is used in a particular construction is either the one in the mem-init-list (if the data member is mentioned there) or else the one initialization declared with the member. The principle is that `try` goes on the initialization that can fail, which means both on the in-class initializer and on any explicit initialization:

```
class C {
    try string s = “xyzzy”;          // (1)
public:
    C() { }                          // 1: s initialized via in-class init ⇒ ‘try’ above
    C(int) : try string{“plugh”} { } // 2: s initialized via mem-init-list ⇒ ‘try’ here
    C(int) : s{} { }                 // 3: string{} is noexcept ⇒ no ‘try’
};
```

### 4.5.2 Compile-time enforcement, static guarantees

> *“Compile and link time enforcement of such rules is ideal in the sense that important classes of nasty run-time errors are completely eliminated (turned into compile time errors). Such enforcement does not carry any run-time costs – one would expect a program using such techniques to be smaller and run faster than the less safe traditional alternative of relying on run-time checking.” — [Koenig 1989]*

It is likely that §4.5.1 `try` expressions and statements should be just an opt-in feature, available if users want to use it (like `override`). If we have them, some teams may elect, in their own code bases, to require `try` on throwing expressions (possibly on exceptions throwing static exceptions), with the help of automated enforcement via a compiler switch or static analysis tool.

For completeness, however, note that we do have an opportunity now (that we will not have later) to consider making `try` required in new code without backward compatibility or noisiness issues. We could require that:

- For a function call where both the caller and callee are declared with a static-exception-specification, the call (or an enclosing) expression must be covered by a `try` if the callee is not `throws(false)`.

For example:

```
int f1() throws;
int f2() throws;

int main() {
    return f1() + f2();                         // error, f1() and f2() could throw
    try return f1() + f2();                     // ok, covers both f1() and f2()
    return try f1() + f2();                     // same
    return try ((try f1()) + (try f2()));       // ok, but redundant
}
```

**Notes** If we had a time machine, we could require `try` to precede every call to an expression that could throw. We cannot do that today without breaking backward compatibility, and because functions that can throw dynamic exceptions are too pervasive (annotating them would be too noisy because it would require annotating the majority of statements).

For functions with static-exception-specifications we don't need the time machine, because no such functions (or callers thereto) exist yet. This gives us an opportunity to require `try` to precede every expression some subexpression of which could throw a static exception, if we so desire.

Some languages, such as Midori, made it a static error to be able to leave function by throwing if it was marked as nonthrowing. We cannot do that with `noexcept` due to backward compatibility; but we can with the proposed `throws(false)` (which is otherwise a synonym for `noexcept`) and apply these rules to give the static compile-time guarantee that such a function cannot exit by throwing. In such a function, the compiler never needs to generate the `noexcept` termination check.

In unconstrained generic code, when the concrete types are not known, it can be unknowable whether an expression throws. In that case, writing `try` can be permitted but not required, and if it is not written and in a given instantiation the expression can throw, the result is a compile-time diagnostic in the function template definition showing the offending type(s) and operations. Concepts can help with this, by stating what functions can throw.

### 4.5.3 `catch` sugars

We can also provide the following syntactic sugars for cleaner and clearer code that reduces ceremonial boilerplate without making the code too obscure or losing information:

- **"`catch{}`":** To reduce ceremony and encourage efficient exception handling, we could consider letting `catch{/*...*/}` with no parameter be a convenience shorthand for `catch(error err){/*...*/}`.
- **"Standalone `catch`" / "less `try` where it's unnecessary boilerplate (blocks)":** To reduce the ceremony and nesting of `try{}` blocks, we could consider allowing standalone `catch` (not following an explicit `try`)

as-if everything between the previous { scope brace and the `catch` were enclosed in a `try{}` block.[15] After all, a major point of exception handling is to keep the normal path clean and direct, and today's ceremony of `try{}` blocks and required extra nesting makes the normal path less clean and direct.

For example:

```
int main() {
    auto result = try g();                          // can grep for exception paths
    cout << "success, result is: " << result;

    catch {                                         // clean syntax for the efficient catch
        cout << "failed, error is: " << err.error();
    }
}
```

In my opinion, the combination of try-expressions/try-statements with this `catch` sugar is doubly desirable, because it lets us write clean, readable code that simultaneously avoids needless ceremony (e.g., the artificial scopes of `try{}` blocks) while adding back actually desirable information that was missing before (`try` to make the exceptional paths visible).

Finally, we could consider helping the common pattern where code wants to handle just one kind of error and propagate the rest. For example:

```
catch (error e) {
    if (e == codespace::something) {
        // do something different, else allow to propagate
    }
    else throw;
}
```

If we allowed naming an `error` *value* in the catch-handler, it would make this common scenario cleaner:

```
catch (codespace::something) {
    // do something different, else allow to propagate
}
```

[15] Note a potential danger: Anytime we consider omitting an explicit scope we run the risk of making the code feel (or actually be) unstructured and undisciplined. In this case, the preceding scope is already explicit in source code (everything from the scope-{ to the `catch`); it's certainly clear to the compiler, and in my opinion likely also clear to the programmer, and if that bears out with some usability testing then it would confirm that the `try{}` boilerplate really is only adding ceremony and nesting, not any actual information or clarity.

### 4.5.4 Discussion

The "expression `try`" aligns well with hand-coded conventions already being adopted in the community, and brings them into the language. For example, it directly replaces the `OUTCOME_TRY` macro and enables writing the same naturally in the language in normal code:

| [Douglas 2018] example | Possible future extension (not proposed) |
|---|---|
| `outcome::result<int> str_multiply2(const string& s, int i) {`<br>`  OUTCOME_TRY (result, convert(s));`<br>`  return result * i;`<br>`}` | `int str_multiply2(const string& s, int i) throws {`<br>`  auto result = try convert(s);`<br>`  return result * i;`<br>`}` |

This feature would help address the concern of some programmers (including the authors of `expected<T,E>`) who have expressed the desire to interleave normal and exceptional control flow in a way that today's exception handling does not support well. Consider again the example from [P0323R3],

```
// P0323R3 expected<int,errc> style: as preferred by some

int caller2(int i, int j) {
   auto e = safe_divide(i, j);
   if (!e)
     switch (e.error().value()) {
     case arithmetic_errc::divide_by_zero: return 0;
     case arithmetic_errc::not_integer_division: return i / j; // ignore
     case arithmetic_errc::integer_divide_overflows: return INT_MIN;
     // Adding a new enum value causes a compiler warning here, forcing code to update.
   }
   return *e;
}
```

Using dynamic exceptions, the code can put the normal control flow together, but it creates a needless extra `try` scope in the normal path, and throwing types is brittle under maintenance if new failure modes are added:

```
// Today's C++ exception style: cleaner, but also more brittle (and more expensive)

int caller2(int i, int j) {
   try {
      return safe_divide(i, j);
   }
   catch(divide_by_zero) { return 0; }
   catch(not_integer_division) { return i / j; } // ignore
   catch(integer_divide_overflows) { return INT_MIN; }
   // Adding a new exception does not cause a compiler warning here, silently incorrect.
}
```

Using this section's proposal, we could write a combination that arguably combines the benefits of both — note that the return value of `safe_divide` is now always a success result in the normal path and always a failure result in the `catch` path, never a conflated success-or-error result that must then separate its own code paths, yet the handling code's structure is still essentially identical to the `expected<double,errc>` style:

| P0323R3 example | Possible future extensions (not proposed) |
|---|---|
| `expected<double, errc> caller(double i, double j, double k) {`<br>`  auto q = safe_divide(j, k);`<br>`  if (q) return i + *q;`<br>`  else return q;`<br>`}` | `double caller(double i, double j, double k) throws {`<br>`  return i + try safe_divide(j, k);`<br>`}` |
| `int caller2(int i, int j) {`<br>`  auto e = safe_divide(i, j);`<br>`  if (!e) {`<br>`  switch (e.error().value()) {`<br>`  case arithmetic_errc::divide_by_zero:`<br>`  return 0;`<br>`  case arithmetic_errc::not_integer_division:`<br>`  return i / j; // ignore`<br>`  case arithmetic_errc::integer_divide_overflows:`<br>`  return INT_MIN;`<br>`  }`<br>`  }`<br>`  return *e;`<br>`}` | `int caller2(int i, int j) {`<br>`  try return safe_divide(i, j);`<br>`  catch {`<br>`  if (err == arithmetic_errc::divide_by_zero)`<br>`  return 0;`<br>`  if (err == arithmetic_errc::not_integer_division)`<br>`  return i / j; // ignore`<br>`  if (err == arithmetic_errc::integer_divide_overflows)`<br>`  return INT_MIN;`<br>`  }`<br>`}` |

## 4.6 Q&A

### 4.6.1 Wouldn't it be better to try to make today's dynamic exception handling more efficient, *instead of* pursuing a different model?

*"The unavoidable price of reliability is simplicity." – C. A. R. Hoare*

We can keep trying that too and continue hoping for a fundamental breakthrough, but we should not use that as a reason to delay investigating a zero-overhead-by-construction model.

We know that today's dynamic exception model has inherent overheads with no known solution for common and important cases such as memory-constrained and/or real-time systems, which require statically boundable space and/or time cost of throwing an exception. For space determinism (memory-constrained systems), I am not aware of any research progress in the past decade. For time determinism (real-time systems), proponents of today's dynamic exceptions expected suitability for real-time systems to be achieved before 2010; but, as noted in §2.3, there have been only a handful of research results (all over a decade old), notably [Gibbs 2005] which unfortunately is not suitable for general use because of its restrictions on the sizes of class hierarchies and reliance on static linking.

We cannot accept that "Standard C++ is not applicable for real-time systems" — that would be an admission of defeat in C++'s core mission as an efficient systems programming language. Therefore we know we cannot live with today's model unchanged.

The overheads described in §2.5 appear to be inherent in the *dynamic* and *non-local* design of today's dynamic exception model: It is fundamental that today's model requires nonlocal properties (notably, that `throw` must perform dynamic allocation and that `catch` by type must perform RTTI), and so there has to be nonlocal overhead in some form (implementations can compete creatively only on where to put it), and there is no known way to achieve space/time determinism for throwing. Therefore, even if we pour millions of dollars more into optimizing today's dynamic exception handling model, we know already that the best-case result would be that the overheads will still be present but somewhat lower (e.g., smaller static tables for less binary size bloat), and throwing will still not be deterministic in either space or time and so exceptions will still be unusable in real-time systems and/or systems without virtual memory.

### 4.6.2 But isn't it true that (a) dynamic exceptions are optimizable, and (b) there are known optimizations that just aren't being implemented?

*"Almost anything is easier to get into than out of." — Agnes Allen*

Yes, but part (a) exposes a fundamental problem just in the way it's phrased, and part (b) could be telling.

The reason (a)'s phrasing exposes a fundamental problem is that it underscores that today's dynamic model *relies* on such optimizations, which is already a sign it fails to be true to C++'s zero-overhead spirit. Contrast:

- **"Pound of cure" [not C++'s ethos].** Today's "out-of-band dynamic exceptions" model is *modeled* as a dynamic feature, *specified* in a way that assumes dynamic overheads, then *relies* on optimization to optimize them away. Relying on the optimizer is a giveaway that we're talking about a "pound of cure" strategy, which is normally avoided by C++ and is a major reason why non-zero-overhead proposals like

C++0x concepts have failed (they were defined in terms of concept maps, and relied on those being optimized away). That strategy is not appropriate for C++; it is the strategy of languages like Java and C# that place other priorities first, and it is precisely because those languages rely on optimization to claw back performance losses that they can never be as efficient as C++ overall, not just in fact but also in principle.

- **"Ounce of prevention" [C++'s ethos].** This paper's proposed "in-band alternate return value" model is *modeled* as a static (stack) feature, *specified* in a way that assumes strictly local stack/register use (designed to share the return channel under the covers), and so does *not rely* on optimizations to optimize overheads away. This kind of strategy is exactly why and how C++ has been uniquely successful as a zero-overhead language: because it (nearly always) avoids incurring overheads in the first place, by construction. Furthermore, it doubles down on C++'s core strength of efficient value semantics.

Could we try to add requirements to today's dynamic exception model, to say things like "local cases must be as efficient as a forward `goto`"? Possibly, but: We have never specified such a thing in the standard, and in this case it would address the problem only in degree, not in kind — it would be a "mandatory optimization" ("required poundage of cure") rather than a correction to fundamentally fix the specification of the operation as a static and local feature, instead of as a dynamic and non-local feature.

For an example of (b), see [Glisse 2013]. The intention is to short-circuit `throw` and `catch` when that is visible to the optimizer (possibly across inlining boundaries), and it could result in performance gains. That it has been mostly ignored and not viewed as a priority is arguably telling. Granted, there could be many reasons why it's not fixed, such as Clang being maintained in part by organizations that themselves ban exceptions and who therefore are disincentivized to optimize them; but if so then the maintainers' being uninterested because they already abandoned exceptions outright is data too.

**Note** Gor Nishanov mentions a specific potential optimization of today's dynamic exception handling model that could alleviate at least the need to have `filesystem`-like dual APIs. Gor writes:

*Consider the case where `catch` and `throw` are in the same function and handler does not call any functions that compiler cannot see the bodies of (to make sure there is no sneaky calls to `current_exception` or `throw;` hiding inside) and the handler does not rethrow. In that case, `throw <expr>;` can place the exception object on a stack and then does a "normal" `goto` into the handler.*

*With that rather straightforward optimization we no longer need to have duplicate APIs as in `<filesystem>`. In our implementation, all the dual APIs are implemented as:*

```
fs_xxx(bla, ec) { ec = fs_xxx_impl(bla); }
fs_xxx(bla)     { if (auto ec = fs_xxx_impl(bla)) throw ec; }
```

*with the exception that we don't throw `ec`, but a fatter object.*

*If [a caller] does local `catch`ing and only uses 'throwing' version of the API, with that simple optimization we will have the codegen identical to a version that request the `ec` and then does handing of the `ec` with an `if` statement. True, the [caller's] `catch` syntax would be more verbose, but, that is something that can be addressed in either exception model.*

### 4.6.3 Can I catch `error` values and dynamic exceptions?

Yes, just write `catch(...)` to catch both or write multiple `catch` blocks for different thrown types as today.

For example:

```
int f() throws { throw std::errc::ENOMEM; }   // report failure as ENOMEM
int g() { throw std::bad_alloc; }             // report failure as bad_alloc
int main() {
    try {
        auto result = f() + g();
    } catch(error err) {                      // catch ‘error’
        /*...*/
    } catch(std::exception const& err) {      // catch ‘std::exception’
        /*...*/
    }
    try {
        auto result = f() + g();
    } catch(...) {
        /*...*/
    }
}
```

To invoke the translation to a common error type, use a function (possibly a lambda):

```
int h() throws {                // bad_alloc → ENOMEM
    return f() + g();
}
int main() {
    try {
        auto result = h();
    } catch(error err) {        // catch ‘error’, incl. translated ‘std::bad_alloc’
        /*...*/
    }
    try { []() throws {         // bad_alloc → ENOMEM
        auto result = f() + g();
    }(); }
    catch(error err) {          // catch ‘error’, incl. translated ‘std::bad_alloc’
        /*...*/
    }
}
```

### 4.6.4 Can `error` carry all the information we want and still be trivially relocatable? For example, `filesystem_error` contains more information, such as source and destination paths

Yes, you can have a `filesystem_error` type that includes additional information and pass it losslessly via `error`. Note that "trivially relocatable" is a surprisingly loose requirement; it is satisfied by many `std::` types, including containers and smart pointers.

### 4.6.5 What about allowing the function to specify the type is throws (e.g., `throws{E}`)?

| | |
|---|---|
| **SG Poll** | The 2018-04-11 SG14 telecon took a poll on pursuing this direction: 4-2-5 (Favor-Neutral-Against). |

*"We want to be able to pass arbitrary, user-defined information … to the point where it is caught. Two suggestions have been made for C++: [1] that an exception be a class, and [2] that an exception be an object. We propose a combination of both." — [Koenig 1989]*

*"[In Midori] Exceptions thrown by a function became part of its signature, just as parameters and return values are." — [Duffy 2016]*

Yes, we can add that extension later if experience shows the need. This paper currently does not propose this because it was more controversial in SG14, but this subsection captures its motivation and design. If we were to adopt this feature, we would reconcile the syntax with conditional `throws(`*`cond`*`)` (see §4.1.4) so that they do not collide.

A possible extension is to allow optionally specifying a custom error code type that can carry additional information until it converts to `error` or another thrown type. There are two main motivations:

- To return a **bigger** or **richer** error type, for example, one that directly carries additional information without type erasure. Because `std::error` can already represent arbitrarily rich information by wrapping an `exception_ptr`, and any performance advantage to doing something different would be expected to be in exceptional (throwing) cases only where optimization is rarely useful, justifying this motivation would require providing examples that demonstrate that throwing an alternative type gives better usability (at `catch` sites) and/or performance by avoiding type erasure.
- To return a **smaller** error type. For example, to return an error type that is smaller than two pointers. Because empirical testing shows that on common platforms there is little measurable performance difference in return-by-value of trivially copyable values of size from 1 to 32 bytes, justifying this motivation would require providing examples of platforms that benefit from throwing a smaller type.

The idea would be additionally permit a function to declare a type-specific **static-exception-specification** of `throws{E}` (or other syntax, just using `{}` for now for convenient discussion without collision with `throws(cond)`) which uses the type `E` instead of `error` and otherwise behaves the same as in the previous section. In this model, a specification of `throws` is syntactic sugar for `throws{error}`. For example:

```
string f() throws{arithmetic_error} {
    if (flip_a_coin()) throw arithmetic_error::something;
    return "xyzzy"s + "plover";    // any dynamic exception is translated to arithmetic_error
}
```

```
string g() throws { return f() + "plugh"; }
                          // any dynamic exception or arithmetic_error is translated to error
```

Or, `std::filesystem` might want to say:

```
directory_iterator& operator++() throws{filesystem_error};
```

For a function declared with a static-exception-specification, if it is declared `throws{E}` then additionally:

- `E` must be a single type name and must be default-constructible, noexcept-movable,[16] and convertible to and from `error`. If `E` is not relocatable, then the implementation needs to call destructors of moved-from `E` objects when returning.
- All the rules in §4.1.1 apply as written, except replacing each mention of `error` with `E`. — Corollary: When a function declared `throws{E1}` calls another declared `throws{E2}` which throws an exception that is not handled, the `E2` object is converted to `E1`; if there is no conversion, then the `E2` object is converted first to `error` and then to `E1`.

The potential benefits of this extension include:

- It enables reporting rich errors while also encouraging handling them locally within a subsystem.
- Experience with Expected [P0323R3] and Midori [Duffy 2016] indicates that being able to handle rich errors locally is desirable and useful, while still propagating more generic errors distantly.

**Note** Some reviewers asked whether we would be able to evolve the `error` type over time. I believe this would enable that, by allowing more than a single well-known type which could be used for future evolutions of (or alternatives to) `error` itself. But I don't think that's sufficient motivation in itself to allow `throws{E}`, as we believe we have enough experience with `error_code` and its evolutions to specify `error` well to be future-proof for decades. The main reasons to allow `throws{E}` is to satisfy the requirements of code that uses Expected today with an error type other than `std::error_code`.

If we pursue this optional generalization, we should address the following concerns:

- Show use cases that demand specifying a custom type `E`: `error` as described herein is efficient, and sufficiently flexible to represent all errors in STL, POSIX, Windows, etc., and can wrap arbitrary richer types and transport them with trivial operations. What uses cases are not covered? Simple beats complex, unless the complexity is essential (to expose an important semantic or performance distinction so that the programmer can control it).
- Address how to discourage large error types: These objects should remain small and avoid heap allocation. One AVX512 register or one cache line is ideal.
- Demonstrate actual benefits from a smaller error type: A smaller type would unlikely give measurable savings over returning a two-pointer-sized `error`. The [Itanium ABI] already optimizes that use case. Niall Douglas reports that when benchmarking options for Boost.Outcome, he found no statistically significant difference for returning up to 32 bytes of data as compared to anything less for an Ivy Bridge CPU, although the answer may be different on other architectures.

[16] Trivially relocatable is encouraged, for efficiency so that it is easier for platform ABIs to return in registers. However, this mechanism does not rely on it, and because the choice of `E` is exposed and left to the function author, it meets the zero-overhead principle of getting only the overheads the author opts into.

### 4.6.6 How will `vector<throwing_type>` work?

The existing `std::vector` works the same as today. If `std::vector` is instantiated with a type that declares a function with a static-exception-specification, then any conditional `noexcept` or `*noexcept*`/`*nothrow*` trait behaves as if the function were declared `noexcept(false)`.

Looking forward, if there is a `std2` where we can change existing entities, then personally I would recommend that `vector` simply mark any function that might throw as `throws`:

- It's simple to teach and learn: A user of `vector` would just write error handling code without worrying about whether for a particular vector specialization error might actually happen or not (as such code has to do today anyway if it is generic code that uses a `vector<T>` for arbitrary `T`).
- It's also efficient: Because `throws` is efficient, there is no longer a performance incentive to perform a conditional calculation to see if for a given specialization it might be able to guarantee it doesn't throw, in order to suppress the exception handling machinery overhead.

### 4.6.7 What about `move_if_noexcept()`? conditional `noexcept`? traits?

They all work the same as today and react as if the function were declared `noexcept(false)`. This proposal is not a breaking change.

### 4.6.8 Will we want more `type_traits` to inspect dynamic vs. static exceptions?

I don't think so. The existing ones, and the `noexcept` operator, continue to work as designed. If anything, I think this proposal's effect on the exception-related type traits will be just to use them less often, not to add new ones, since many uses of the traits are motivated by wanting to elide the overhead of today's exception handling (notably to write conditional `noexcept` tests).

### 4.6.9 But people who can't use exceptions typically can't use dynamic memory anyway, right? So there's no benefit in trying to help them use exceptions on their own.

No. That may be true of some environments, but not of most. For example, the Windows kernel allows dynamic memory allocation (after all, it owns and manages the heap) but currently bans exceptions.

### 4.6.10 How does the `try`-expression in §4.5 compose with `co_await`?

§4.5 suggests allowing `try` on any statement or expression (including subexpressions) that could throw. In [O'Dwyer 2018], Arthur O'Dwyer makes the excellent point that "coloring" like `try` and `co_await` should be composable, and used only for essential qualities. His example is:

```
auto result = try (co_await bar()).value();
```

Recall this similar example from §4.5, which proposes allowing `try` on an expression or statement any subexpression of which can throw, and indicates that mental model implies the following two lines are equivalent:

```
try return "xyzzy"s + "plover";          // ok, covers both ""s and +
return try "xyzzy"s + "plover";          // same
```

For the same reasons, the following are also equivalent ways to call an asynchronous function that can fail:

```
int result = co_await try foo();
int result = try co_await foo();
```

Incidentally, this is a counterexample that argues against the suggestion in [D0973R0] to co-opt `co_await` to mean what §4.5 suggests be a `try`-expression: If `co_await` can only mean one thing at a time, and one abuses it so that sometimes that thing is error handling, then how do you express invoking an operation that both can report an error and is asynchronous? (There are many other reasons `co_await` should not be co-opted for this meaning, including that error handling should be distinct from all normal control flow.)

### 4.6.11 Wouldn't it be good to coordinate this ABI extension with C (WG14)?

Yes, and that is in progress.

This paper proposes extending the C++ ABI with essentially an extended calling convention. Review feedback has pointed out that we have an even broader opportunity here to do something that helps C callers of C++ code, and helps C/C++ compatibility, if C were to pursue a compatible extension. One result would be the ability to throw exceptions from C++→C→C++ while being type-accurate (the exception's type is preserved) and correct (C code can respond correctly because it understands it is an error, even if it may not understand the error's type).

It could simplify our C++ implementation engineering if C functions could gain the ability to return A-or-B values, so for example:

```
_Either(int, std_error) somefunc(int a) {
    return a > 5 ? _Expected(a) : _Unexpected(a);
}
// ...
_Either(int, std_error) ret = somefunc(a);
if(ret)
    printf("%d\n", ret.expected);
else
    printf("%f\n", ret.unexpected);
```

Here `_Either` would be pure compiler magic, like `va_list`, not a type. Under the hood, functions would return with some CPU flag (e.g., carry) set if it were a right value, clear if it were a left value. An alternative is setting a bit in the thread environment block which probably is much better, as it avoids a calling convention break, though potentially at the cost of using thread local storage. It would be up to the platform vendor what to choose (this is an ABI extension), but the key is that C and platforms' C-level calling conventions would be extended to understand the proposed union-like return of values from functions to use the same return channel storage, with a bit to say what the storage means.

In C++, functions marked `throws` would, in C terms, return `_Either(T, std::error)` and this is how the exception throw would be unwound up the stack, by the C++ compiler testing after every `throws` function call if an unexpected `std::error` value was returned, and if so returning that `std::error` up to caller.

Thus, these C++ functions:

```
extern "C++" int do_something(double) throws;
extern "C++" double do_something_else(int) noexcept;
```

would in C become:

```
extern _Either(int, std_error) _Z3do_somethingd(double);
extern double _Z3do_something_elsei(int);
```

A major benefit of (also) doing it at the C level is that C is the lingua franca portable platform language that all other languages bind to. Therefore this facility would help integrate C not only with C++ error handling, but also with Rust, Swift, and many other languages that can speak C and also return A-or-B from functions as their main error handling implementation. The carrot for WG14 would be that C facility would benefit all C speaking languages, and help C be more useful in its key role as a glue language. A notable gain is that C could now receive exception throws from C++, and propagate them, perhaps even convert them for other languages.

Further, there are benefits for embedded use of C:

- On many common microcontrollers, this allows a more efficient implementation of any function that currently produces/propagates errors using return plus an out parameter for the return value. Adding parameters to a function is expensive, more expensive than using a flag.
- This provides a usable replacement for `errno` which is not an option in interrupt service routines or preemptive run-to-completion kernels (either adds ISR latency or potentially yields false results).

In C++, I would like to see group discussion of this question and get direction of SG14 and EWG regarding their interest in this proposal if C does, or does not, also make a coordinated ABI call convention extension with C++. Is coordination with C necessary for this proposal to be interesting for C++, or should this proposal be explored as of potential interest for a future C++ regardless of what C does?

### 4.6.12 Would language level variants and pattern matching lead to a different error handling style and change the design and goal of this proposal?

This proposal is agnostic to a language variant, and would benefit from pattern matching in `catch` clauses.

A variant type has the same effect on this proposal whether it's language-level or a library. For example, as described in the §2.4 Note, we have a "return a variant for error reporting" idiom before us already in one of `expected<T,U>`'s two major use cases:

- `expected<T,U>` where `T` and `U` are alternate success results: Both types are naturally dealt with using normal control flow, we should prefer using `variant<T,U>` today, and we can use a language-level variant too if we get one. But this case is not about error handling, so it doesn't affect this proposal.
- `expected<T,E>` where `E` really represents an error: This proposal already blesses that with language integration into the return channel (as efficient as a language-level variant could do) plus language support for keeping the error-handling paths distinct and automatically propagating the errors, which is equally an improvement whether the returned variant is a language or library feature.

Pattern matching will be useful to categorize errors within `catch` clauses (see §4.1.6 Notes) just as it's useful anywhere else. What is essential, however, is to keep normal and error handling control paths separate, with the latter entirely in `catch{}` blocks. We want to get away from the idiom of using normal control flow constructs for error handling which interleaves the normal and error code paths — and "normal control flow constructs" means pattern `match` just as much as it means `if` and `switch`. Just as we don't want to use `if` and `switch` to select error handling blocks (e.g., `if(result.is_error(){}`, or `switch(result){` *`/*some cases are error cases, or cases are successes and default is the error case*/`* `}`), the same is true for future pattern matching (e.g., `match(result){` *`/*some cases are error cases, or cases are successes and default is the error case*/`* `}`).

### 4.6.13 What about projects that assume that all exceptions inherit from `std::exception`?

Some projects have made the assumption that all exceptions inherit from `std::exception`, and that anything else is an obvious developer mistake enforced by `catch(...){assert(false);}` guards. However, because C++ dynamic exceptions allow throwing any type, such projects still have to be aware of, and work with, called code that does not follow it. That code already does not work with arbitrary third-party libraries without adding some awareness of non-`std::exception`-derived exception objects. Some C++ libraries provide their own competing exception base class types; for example, Boost provides `boost::exception` that is "designed to be used as a universal base for user-defined exception types"[Dotchevski 2009]. Also, C++ books have long thrown other types (such as integers) in their examples.

### 4.6.14 Should we consider overloading on `throws`?

Probably not. That would need strong justification, and address the problems raised when we considered proposals to allow overloading on `noexcept`.

Some reviewers have suggested that we could allow overloading on "undecorated" and `throws` (e.g., `void f();` and `void f() throws;`) with the meaning that, as a final tie-break, a function with/without a static-exception-specification will prefer invoking another one with/without a static-exception-specification. This means `throws` would also become part of the function's type as with `noexcept`, but stronger because we do not allow overloading on `noexcept`(see [P0012R1]). This would be primarily useful to optimize the translation of dynamic exceptions to `error`s in common functions. For example, in this proposal (and if some form of §4.3 is not adopted), a function with a static-exception-specification that invokes today's existing `operator new` and doesn't handle a `bad_alloc` would get it automatically translated to `std::errc::ENOMEM`. A quality implementation that inlines the call to `operator new` could elide the dynamic exception, but that relies on an optimization. If we supported overloading on `throws`, then we could additionally provide a (non-placement) overload of `operator new` that is decorated with `throws`, and it will be used by `throws` functions to guarantee no dynamic exception ever happens (whether the call is inlined or not) while leaving all existing uses unchanged (all existing code still uses the existing `operator new`).

My current view is that overloading on `throws` adds complexity that would need a strong set of compelling examples to justify, and it would need to address the good reasons why we do not overload on `noexcept` today and have not pursued proposals that suggested allowing it.

### 4.6.15 What if this function is calling others that could throw different things?

That situation already exists today: This function might call a variety of other functions, each of which may throw different things. This function just separately decides what errors it will communicate (that types it may throw, or opt into `throws`), and internally it catches whatever the functions it calls may throw, as today:

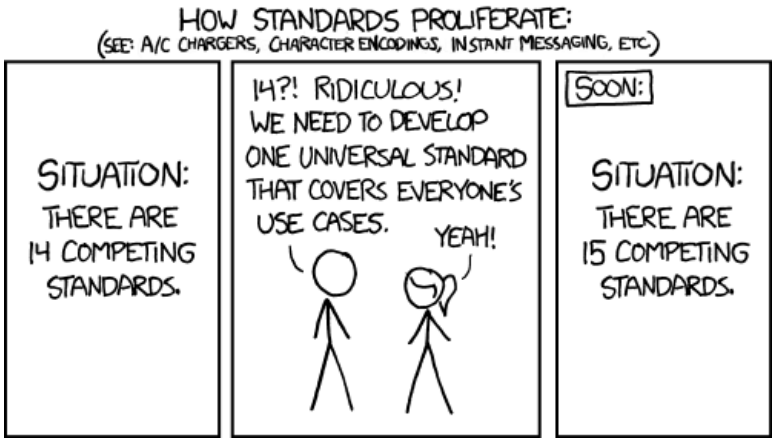
- If it calls something that can emit a `filesystem_error`, it should `catch(filesystem_error&)` (or a base thereof) if it wants to handle that.
- If it calls something that can emit a `bad_cast`, it should `catch(bad_cast&)` if it wants to handle that.
- ...
- (new) The new thing is that if it calls something that can emit a `std::error`, it should `catch(std::error)` if it wants to handle it. There's nothing new except that now you can catch a `std::error`, and you can (and should prefer to) catch that one by value.
- `catch(…)` continues to catch anything.

# 5 Dangers, and "what to learn in a TS"

*"Adding a mechanism to 'solve the error-signaling problem' is more likely to add yet-another-alternative to this mess, than to solve it." — [P0976]*

Here are two major groups of dangers we must address (e.g., in a TS) before considering adopting this proposal into a draft standard, with sample hypotheses that if validated would address them.

**(1) Ending up with N+1 alternatives**. To avoid xkcd 927 (see right), we must validate that this proposal is unlikely to increase the number of alternative C++ error handling methods. One way to do that is to demonstrate it can replace at least one of the existing alternatives already in development and use (e.g., Outcome, Expected), for example if the proposers of one of those documents that this proposal can be viewed as a direct generalization that can subsume the existing alternative. That would demonstrate that this proposal at least does not increase the number of alternatives.




**(2) Replacing known problems with something having unknown problems.** We must create an experimental implementation and try it against real projects to validate its efficiency and usability, and to identify its currently-unknown problems. Also, we reduce this risk by documenting how the proposal directly relates to existing practice with known qualities in C++ and other languages. Note two specific cases of the latter problem:

- **The danger that this proposal might encourage excessive local propagation (e.g., a pattern of writing otherwise-unnecessary code to `catch` from a `throws{E}` before `E` is converted to `error`).** We must validate that projects using an experimental implementation still throw and catch exceptions at whatever distance is normal, without resorting to otherwise-undesirable local catches before a more specific error type is translated to `error`.
- **The danger that we cause users to reinvent a form of RTTI.** We must validate that projects do not use these facilities to catch `error`s at higher code levels in a way where they then perform run-time type/categorization efforts that are tantamount to duplicating RTTI in a way that falls into today's problems of RTTI (see Note in §2.5).

*"Haskell does something even cooler and gives the illusion of exception handling while still using error values and local control flow" — [Duffy 2016]*

Finally, we must not mimic other languages' choices just because things appeared to have worked out well for them. Languages are different, even languages as similar as C and C++, and what works well in one language does not necessarily work well or at all in another. So, although the rationale and core design of this proposal arrives at similar conclusions and basic designs as existing practice in C++ (e.g., `std::error_code`, Expected, Outcome) and other modern languages (e.g., Midori, and in parts CLU, Haskell, Rust, and Swift), and so benefits from experience in those languages, it is not motivated by mimicry and we cannot rely only on experience in those languages. This is a proposed feature for C++, and it has to be prototyped and tested in a C++ compiler and with C++ code to get relevant data about its actual performance and usability.

# 6 Bibliography

**Note** Many of these references were published in the past year. C++ error handling continues to consume copious committee and community time wrestling with unresolved issues, over a quarter of a century after C++ exceptions were designed and implemented.


[Abseil] Abseil Common Libraries. (Google, 2018).

[Alper 2019] "How many values are errors in Go?" (2019-06-11).

[babb] H. Sutter and M. Clow. "babb: bad_alloc Behaving Badly" (GitHub, 2019).

[Bay 2018] C. Bay. "Boost.Outcome.v2 Review Report" (Boost, 2018-02-05).

[Brooks 1975] F. Brooks. *The Mythical Man-Month* (Addison-Wesley, 1975).

[Cox 2018] R. Cox. "Go 2 Drafts announcement" (YouTube, 2018-08-28).

[C# OOM] "OutOfMemoryException Class" (Microsoft).

[Dechev 2008] D. Dechev, R. Mahapatra, B. Stroustrup, D. Wagner. "C++ Dynamic Cast in Autonomous Space Systems" (IEEE ISORC 2008, 2008-05).

[Dechev 2008a] D. Dechev, R. Mahapatra, B. Stroustrup. "Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems" (Journal of Computing Science and Engineering (JCSE), 2:4, 2008-12).

[Dotchevski 2009] E. Dotchevski. "Boost Exception" (boost.org, 2009).

[Douglas 2018] N. Douglas. "Outcome 2.0 Tutorial" (2018).

[Douglas 2018a] N. Douglas. "Reference implementation for proposed SG14 `status_code` (`<system_error2>`) in C++ 11" (2018).

[Douglas 2018b] N. Douglas. "Header file `error.hpp`" — A prototype implementation of this paper's `error` (2018-03-03).

[Duffy 2015] J. Duffy. "Safe native code" (*Joe Duffy's Blog*, 2015-12-19).

[Duffy 2016] J. Duffy. "The error model" (*Joe Duffy's Blog*, 2016-12-07). Describes Midori's error handling model.

[Embedded C++] "Rationale for the Embedded C++ specification" (Embedded C++ Technical Committee, 1998-11-20).

[Edge] Microsoft Edge developer documentation.

[Filesystem v3] B. Dawes. "Filesystem Library Version 3" (Boost, 2015-10-25).

[Gibbs 2005] M. Gibbs, B. Stroustrup. "Fast dynamic casting" (Lockheed Martin & Texas A&M University collaboration; *Software—Practice and Experience* 2006; 36:139–156). Published online 2005-09-15 in Wiley InterScience (www.interscience.wiley.com).

[Glisse 2013] M. Glisse. "Remove `throw` when we can see the `catch`" (LLVM bug 17467, 2013-10-03).

[Go OOM] R. Gooch. "Proposal: Make it possible to catch failed memory allocations" (Golang repo, 2016-01-30).

[Goodenough 1975] J. B. Goodenough. "Exception handling: Issues and a proposed notation" (CACM, 18(12), 1975-12).

[Gorman 2007] M. Gorman. "Understanding the Linux virtual memory manager" (2007-07-09).

[GSG] *Google C++ Style Guide* (Google).

[GSL] C++ Core Guidelines' Guidelines Support Library.

[Haskell 2009] (user "Lemming") "Error vs. Exception [in Haskell]" (wiki.haskell.org, 2009-12-07).

[Ignatchenko 2018] S. Ignatchenko. "App-level Developer on std::error Exceptions Proposal for C++. Part I. The Good" (blog post, 2018-05-23).

[Ignatchenko 2018a] S. Ignatchenko. "App-level Developer on std::error Exceptions Proposal for C++. Part II. The Discussion" (blog post, 2018-05-30).

[ISO 18015:2004] *Technical Report on C++ Performance* (ISO/IEC TR 18015:2004(E), 20015-06-15). Also available at http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.

[Itanium ABI] *Itanium C++ ABI* (GitHub, 2017-03-14).

[JSF++ 2005] *Joint Strike Fighter Air Vehicle C++ Coding Standards* (Lockheed Martin, 2005).

[Klabnik 2017] S. Klabnik, C. Nichols. "Recoverable errors with `Result`" (The Rust Programming Language, 2nd ed., 2017-10-28)

[Koenig 1989] A. Koenig, B. Stroustrup. "Exception Handling for C++" (Proc. C++ at Work conference, 1989-11. Revision published in Proc. USENIX C++ Conference , 1990-04, and Journal of Object Oriented Programming, 1990-07).

[Koenig 1996] A. Koenig. "When memory runs low" (*C++ Report*, 8(6), June 1996). Scanned and reproduced by Michael Marcin (thank you Michael!).

[Landley] R. Landley. "Linux Memory FAQ" (date unknown)

[Lee 2015] B. Lee, C. Song, T. Kim, W. Lee. "Type Casting Verification: Stopping an Emerging Attack Vector" (24th USENIX Security Symposium, 2016-08-12). Video.

[Lippincott 2016] L. Lippincott. "What is the basic interface?" (CppCon 2016, 2016-09-19). Video.

[Liskov 1979] B. Liskov, A. Snyder. "Exception handling in CLU" (IEEE Transactions in Software Engineering, 1979).

[Liskov 1992] B. Liskov. "A history of CLU" (1992).

[LKA 2018] "Overcommit accounting" (The Linux Kernel Archives, 2018-05-02).

[LWG 3013] T. Song et al. "`(recursive_)directory_iterator` construction and traversal should not be `noexcept`" (WG21 LWG Issues List, retrieved 2018-03-24).

[LWG 3014] T. Song et al. "More `noexcept` issues with filesystem operations" (WG21 LWG Issues List, retrieved 2018-03-24).

[LWG 3038] B. O'Neal III. "`polymorphic_allocator::allocate` should not allow integer overflow to create vulnerabilities." (WG21 LWG Issues List, retrieved 2019-06-11).

[Maimone 2014] M. Maimone. "C++ on Mars: Incorporating C++ into Mars Rover Flight Software" (CppCon, 2014-09-10).

[McCall 2016] J. McCall et al. "Error handling rationale and proposal" for Swift. (Swift GitHub repo, 2016).

[Müller 2017] J. Müller. "Exceptions vs. `expected`: Let's find a compromise" (Jonathan Müller's blog, 2017-12-04).

[N2271] P. Pedriana. "EASTL: Electronic Arts Standard Template Library" (WG21 paper, 2007-04-27).

[N3051] D. Gregor. "Deprecating exception specifications" (WG21 paper, 2010-03-12).

[N3239] B. Dawes. "Filesystem Library Update for TR2 (Preliminary)" (WG21 paper, 2011-02-25).

[O'Dwyer 2017] A. O'Dwyer. "`dynamic_cast` From Scratch" (CppCon 2017, 2017-09-26). Talk slides. Source code.

[O'Dwyer 2018] A. O'Dwyer. "Async/await, and coloring schemes in general" (Blog post, 2018-03-16). The example in "Coloring schemes don't stack," which uses a `try`-expression, was motivated in part by a draft of this paper.

[O'Dwyer 2018a] A. O'Dwyer. "The best type traits… that C++ doesn't have (yet)" (C++ Now 2018, 2018-05-08).

[O'Dwyer 2018b] A. O'Dwyer. "`[[trivial_abi]]` 101" (Blog post, 2018-05-02).

[P0012R1] J. Maurer. "Make exception specifications be part of the type system, version 5" (WG21 paper, 2015-10-22).

[P0068R0] A. Tomazos. "Proposal of `[[unused]]`, `[[nodiscard]]` and `[[fallthrough]]` attributes" (WG21 paper, 2015-09-03).

[P0132R0] V. Voutilainen. "Non-throwing container operations" (WG21 paper, 2015-09-27).

[P0323R3] V. Botet, JF Bastien. "Utility class to represent `expected` object" (WG21 paper, 2017-10-15). (Current design paper for `expected<T,E>`.)

[P0323R5] V. Botet, JF Bastien. "`std::expected`" (WG21 paper, 2018-02-08). (Current wording paper for `expected<T,E>`.)

[P0364R0] M. Wong, S. Srivastava, S. Middleditch, P. Roy. "Report on Exception Handling Lite (Disappointment) from SG14… or, How I learned to stop worrying and love the Exception Handling" (WG21 paper, 2016-05-23).

[P0380R1] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup. "A Contract Design" (WG21 paper, 2016-07-11).

[P0542R3] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup. "Support for contract based programming in C++" (WG21 paper, 2018-02-12).

[P0762R0] N. Douglas. "Concerns about `expected<T, E>` from the Boost.Outcome peer review" (WG21 paper, 2017-10-15).

[P0779R0] N. Douglas. "Proposing `operator try()`" (WG21 paper, 2017-10-15).

[P0788R2] W. Brown. "Standard library specification in a concepts and contracts world" (WG21 paper, 2018-02-03).

[P0824R1] A. O'Dwyer, C. Bay, O. Holmes, M. Wong, N. Douglas. "Summary of SG14 discussion on `<system_error>`" (WG21 paper, 2018-02-05).

[P0829R2] B. Craig. "Freestanding proposal" (WG21 paper, 2018-05-07).

[P0939R0] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong. "Direction for ISO C++" (WG21 paper, 2018-02-10).

[P0941R0] V. Voutilainen. "Integrating feature-test macros into the C++ WD" (WG21 paper, 2018-05-04).

[P0973R0] G. Romer, J. Dennett. "Coroutines TS Use Cases and Design Issues" (WG21 paper, 2018-03-23).

[P0976R0] B. Stroustrup. "The evils of paradigms, or Beware of one-solution-fits-all thinking" (WG21 paper, 2018-03-06).

[P1028R0] N. Douglas. "SG14 `status_code` and standard `error` object for P0709 Zero-overhead deterministic exceptions" (WG21 paper, 2018-04-28).

[P1029R0] N. Douglas. "SG14 `[[move_relocates]]`" (WG21 paper, 2018-05-01)

[Rust OOM] "Allocation failure should panic" (Rust-lang.org, 2017-03-01). Discusses OOM panic behavior, and points to a working group that is considering allocators that can return `Result` (these appear to all be `unsafe` at the moment, and the working group appears to be still ongoing).

[SC++F 2018] "Results summary: C++ Foundation Developer Survey 'Lite', 2018-02" (Standard C++ Foundation, 2018-03-07).

[Schilling 1998] J. Schilling. "Optimizing away C++ exception handling" (ACM SIGPLAN Notices, 33(8), 1998-08).

[StackOverflow 2010] B. Larsen. "Why does my program occasionally segfault when out of memory rather than throwing bad_alloc?" (StackOverflow, 2010-04-02). Even when using `ulimit`.

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[Stroustrup 2004] B. Stroustrup. “Abstraction and the C++ machine model” (ICESS ’04, 2004).

[Sutter 1997] H. Sutter. “Code Complexity, Part I” (Blog post, 1997-09-14). An updated version appeared as Item 18 of *Exceptional C++* (Addison-Wesley, 2000).

[Sutter 2001] H. Sutter “To new, perchance to throw” (*C/C++ Users Journal*, 19(5), May 2001).

[Sutter 2002] H. Sutter. “A pragmatic look at exception specifications.” (C*/C++ Users Journal*, 20(7), 2002-07.)

[Squires 2017] J. Squires, JP Simard. “Error handling in Swift: A history” (*Swift Unwrapped* podcast, 2017-06-19).

[Swift OOM] Discussion thread (2016-04-14). Comment by Chris Lattner: “Swift’s policy on memory allocation failure is that fixed-size object allocation is considered to be a runtime failure if it cannot be handled. OTOH, APIs that can take a variable and arbitrarily large amount to allocate should be failable.”

# Appendix: Illustrating “stack” exception allocation (Windows)

Below is a simple test program to measure the stack overhead of exception handling using the Windows model. Recall from §2.5.1, point (2), that the Windows exception handling model attempts to avoid having to heap-allocate exception objects by optimizing them to be physically on the stack, but not with usual stack semantics.

This test program has only two short functions, one of which calls the other three times. The test measures the stack impact of:

- “Current” vs. “Proposed”: reporting errors via exceptions (“Current”) vs. a `union{result; error;}` + `bool` (“Proposed”, a naïve implementation of this proposal written by hand; if this proposal were implemented, the source code for both tests would be the same, `try`/`catch`).
- Frame-based vs. table-based: in release builds for x86 (no tables) and x64 (table-based).

The code follows at the end. The following prints the memory offsets of various local and exception/error objects relative to a local stack variable in the topmost caller. The runs were on Release x86 and Release x64 builds using Visual Studio 2017 15.7, which use frame-based and table-based implementations, respectively, of today’s dynamic exception handling.

Release x86 (frame-based)

| | ---- Stack offsets ---- | |
|---|---|---|
| | Current | Proposed |
| test.c: | -56 | 1 |
| test2.c: | -120 | 2 |
| test2.exception: | -139 | -51 |
| test.exception: | -139 | -67 |
| test2.c: | -2,280 | 3 |
| test2.exception: | -2,299 | -35 |
| test.exception2: | -2,299 | -67 |
| test2.c: | -4,424 | 4 |
| test2.exception: | -4,443 | -19 |
| test.exception3: | -4,443 | -67 |
| main.exception: | -139 | 5 |

Release x64 (table-based)

| | ---- Stack offsets ---- | |
|---|---|---|
| | Current | Proposed |
| test.c: | -96 | 1 |
| test2.c: | -216 | 2 |
| test2.exception: | -240 | -72 |
| test.exception: | -240 | -96 |
| test2.c: | -19,000 | 3 |
| test2.exception: | -19,024 | -48 |
| test.exception2: | -19,024 | -96 |
| test2.c: | -37,704 | 4 |
| test2.exception: | -37,728 | -24 |
| test.exception3: | -37,728 | -96 |
| main.exception: | -240 | 8 |

The offsets show how in the “Current” (today’s exception handling) case, not only are the stacks fattened up by the `try`/`catch` machinery in both x86 and x64, but the effect mentioned in §2.5.1 indeed ‘pins’ the already-destroyed-and-otherwise-ready-to-reclaim stack while each exception is handled. The mockup of this proposal flattens the stack, including showing reuse of the same storage for `test.exception`, `test.exception2`, and `test.exception3` (here by hand coding, but I expect existing optimizers to routinely do it if the two lines marked `// can reuse result` were preceded with `auto` to declare new nested variables, and we weren’t taking their addresses to print them).

**Note** On non-Windows platforms, using gcc (Wandbox) and Clang (Wandbox), the three exceptions similarly create three distinct allocations, but on the heap with addresses far from the stack. Still, they are required by the current model to be three distinct addresses there too, and three distinct heap allocations modulo heroic optimizations.

Finally, note that this measures only stack space savings on Windows platforms, and does not attempt to measure the other primary savings this proposal aims to achieve (e.g., elimination of global state such as tables).

## Sample code

```
#include <vector>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;


//=============================================================================
//     Helpers and instrumentation

//     Quick-and-dirty GSL (avoiding assert because I'm testing only release builds)
namespace gsl {
       void Expects(bool b) { if (!b) abort(); }
       using index = ptrdiff_t;
}

//     Counters
vector<string> labels;
vector<int>    run1, run2;
vector<int>*   prun  = nullptr;
char*          pbase = nullptr;

//     Record the address of a given object relative to the current test stack base
template <class T>
void addr(T& p, const char* msg) {
       gsl::Expects(prun != nullptr && (prun == &run1 || prun == &run2));
       if (prun == &run1) labels.push_back(msg);
       else gsl::Expects(labels[run2.size()] == msg);
       prun->push_back(&(char&)p - pbase);
}

//     Print results
void print_results() {
       gsl::Expects(labels.size() == run1.size());
       gsl::Expects(run1.size() == run2.size());

       cout.imbue(std::locale(""));
       cout << setw(45) << "---- Stack offsets ----" << endl;
       cout << setw(34) << "Current";
       cout << setw(11) << "Proposed" << endl;

       for (gsl::index i = 0; i < (gsl::index)labels.size(); ++i) {
              cout << setw(20) << labels[i] << ":  ";
              cout << setw(11) << run1[i];
              cout << setw(11) << run2[i] << endl;
       }
}


//=============================================================================
//     Test types

struct success { int* _; int32_t __; };
struct error   { int* _; int32_t __; };
struct folded  { union { success s; error e; } u; bool b; };
```

```
//=============================================================================
//     Test for today's EH

namespace Current {

	success test2() {
		char c;
		addr(c, "test2.c");

		try {
			throw exception();
		}
		catch (exception& e)
		{
			addr(e, "test2.exception");
			throw;
		}

		return success();
	}

	success test() {
		char c;
		addr(c, "test.c");
		success result;
		try {
			result = test2();
		}
		catch (exception& e)
		{
			addr(e, "test.exception");
			try {
				result = test2();
			}
			catch (exception& e)
			{
				addr(e, "test.exception2");
				try {
					result = test2();
				}
				catch (exception& e)
				{
					addr(e, "test.exception3");
				}
			}
			throw;
		}

		return result;
	}


	int main()
	{
		prun = &run1;

		char base;
		pbase = &base;

		try {
```

```
                auto result = test();
                return result._ != nullptr;
        }
        catch (exception& e)
        {
                addr(e, "main.exception");
                return 0;
        }
    }

}

//=============================================================================
//     Test for proposed EH

namespace Proposed {

    folded test2() {
        char c;
        addr(c, "test2.c");
        folded result;

        //try {
                result.u.e = error();
                result.b = false;
        //}
        //catch (exception& e)
        //{
                addr(result.u.e, "test2.exception");
                return result;
        //}

        result.u.s = success();
        result.b = true;
        return result;
    }

    folded test() {
        char c;
        addr(c, "test.c");
        folded result;
        //try {
                result = test2();
        //}
        //catch (exception& e)
        if (!result.b)
        {
                addr(result.u.e, "test.exception");
                //try {
                        result = test2();           // can reuse 'result'
                //}
                //catch (exception& e)
                if (!result.b)
                {
                        addr(result.u.e, "test.exception2");
                        //try {
                                result = test2();    // can reuse 'result'
                        //}
                        //catch (exception& e)
                        if (!result.b)
```

```
                        {
                                addr(result.u.e, "test.exception3");
                        }
                }
        }

        return result;
}


int main()
{
        prun = &run2;

        char base;
        pbase = &base;

        //try {
                auto result = test();
                if (result.b) return result.b == true;
        //}
        //catch (exception& e)
        if (!result.b)
        {
                addr(result.u.e, "main.exception");
                return 0;
        }
}

}

//=============================================================================

int main() {
        Current::main();
        Proposed::main();
        print_results();
}
```