Doc. No. P1947
Date: 2019-11-18
Audience: EWG and LEWG
Reply to: Bjarne Stroustrup (bs@ms.com)

C++ exceptions and alternatives

Bjarne Stroustrup

Abstract

Exceptions are necessary, as are error codes. This paper discusses problems, costs, and risks involved in replacing C++ exceptions with alternatives (e.g., so-called “deterministic exceptions”). It recommends against adding new error-handling mechanisms to C++. It encourages careful studies of improved error-handling styles and techniques. It encourages study of potential improvements in the implementation of (existing) C++ exceptions.

The “deterministic exception” proposals tease us with hopes of healing the schism in the C++ community caused by needs for no-exception compiler options and even of finally healing the C/C++ schism. To many, massive efficiency improvements seem easily obtainable. Others see a promise of safer and more readable code. Unfortunately, those great hopes are most unlikely to become fulfilled. Yet-another error-handling mechanism/style will fracture the community further because the existing mechanisms and styles won’t disapear, efficiency improvements will be patchy and not benefit anywhere near all users, the work on new mechanisms will delay much-needed work on optimization of existing facilities and on novel other features. Even the promise of “determinism” will be only partially met.

I don’t refer to specifics of proposals (e.g., [Sutter] and [Douglas]). I don’t think the time for language details has come; maybe it will never come. I think the whole direction is wrong. We should focus on improving the C++ exceptions and on using them well.

Introduction

This is a negative paper because it opposes proposals without suggesting specific improvements or novel alternatives. I hate writing such; it does not seem constructive, but novel features can do harm. Fundamentally, the burden of proof is on the proposers. Here, I outline some of that burden. In the case of “deterministic exceptions” and other suggested alternatives to C++ exceptions, I don’t think that burden is remotely met. It is natural to be enthusiastic about a new proposal and – especially in presentations – to emphasize its strength, but in the context of a standards proposal equal weight, or more, should be placed on the evaluation of disadvantages and potential problems. ***First do no harm!*** The C++ exceptions are not perfect – that was known from the onset and very few major language features are perfect – but over decades exceptions have served millions of people well in the ways that they were designed to do. They still do.

Overselling alternative error-handling schemes can do harm by raising unrealistic hopes and by scaring people away from reasonable uses of C++ exceptions (and RAII) and from C++ itself by overemphasizing their negatives.

History

The origins of exception handling lie in the problems experienced managing a variety of error-handling approaches, such as C's **errno**, error-states, callbacks, return codes, and return objects. In addition, it was observed that there were no really good approaches to reporting errors detected in constructors and in operators.

I and others were very aware of the literature on exceptions and error handling stretching back to Goodenough's pioneering work. Before exceptions were adopted into C++, many standards meetings had sessions devoted to the topic and industry experts from outside the committee were consulted. Implementation alternatives were examined, and ideals were articulated. A relatively brief description can be found in D&E.

The integration of exception handling with constructor/destructor resource managements in the form of RAII is one of the major achievements of C++ – still not fully appreciated by people relying on error-code based error handling.

Error codes and exceptions

There have always been applications for which the use of exceptions was unsuitable. Examples Include

- Systems where the memory is so limited that the run-time support needed for exception handling crowds out needed application functionality.
- Hard-real time systems where the tool chains cannot guarantee prompt response after a throw. That is not an inherent language problem.
- Systems relying on multiple unreliable computers so that immediate crash-and-restart is a reasonable (and almost necessary) way of dealing with errors that cannot be handled locally.

Consequently, most C++ implementations always had a no-exceptions switch. On the other hand, there are problems for which error codes provide no good solution:

- Constructor failures – there is no return value (except the constructed object itself). Pure RAII must then be replaced by explicit checks on an object's state plus some handler code; e.g., **Gadget g {args}; if (!g.valid()) handle_construction_error(g);**
- Operators – there is no place to return an error indicator from an operator, e.g., from **++**, *****, or **->**. You will have to use non-local error indicators or live with an impoverished notation, e.g., **multiply(add(a,b),c)** rather than **(a+b)*c**.
- Callbacks – where the function using the callback should be able to invoke functions with a wide variety of possible errors (often, callbacks are lambdas).
- Non-C++ code – there is no way to propagate an error-code through a function that is not C++ and hasn't been written specifically to deal with error codes.

- People forgetting to test a return code – there are clever schemes to try to ensure that error-codes are checked consistently, but they are either incomplete or rely on exceptions or termination once a failure to check is detected.

The zero-overhead principle

It has been repeatedly stated that C++ exceptions violate the zero-overhead principle. This was of course discussed at the time of the exception design. Unsurprisingly, the answer depends on how you interpret the zero-overhead principle. For example, a virtual function call is slower than an "ordinary" function call and involves memory for a vtbl. It is obviously not zero-overhead compared to a simple function call. However, if you want to do run-time selection among an open set of alternatives the vtbl approach is close to optimal. Zero-overhead is not zero-cost; it is zero-overhead compared to roughly equivalent functionality.

Similarly, the exception mechanism was compared to alternatives in a program that needed to catch all errors and then either recover or terminate.

- A program that responds to any error that cannot be handled locally by some form of immediate termination (e.g., reporting to another computer and going into a safe state) was considered outside the scope of the exception mechanism. And thus, not a violation.
- Schemes that consider an exception throw as just an alternative return path was considered different: ***not C++ exception handling***. For simple alternative return values, error codes were considered the answer. Thus, simply demonstrating that a **throw** is slower than a **return** does not demonstrate a violation.
- The kind of comparison that was considered fair was between consistently using error codes and using exceptions for rare (exceptional) errors. In that case, the performance comparison was between an exception handling scheme (table based or otherwise) and a systematic combinations of error codes (e.g., (error-code, value) pairs) and their testing.
- The use of inheritance to represent clusters of open sets of exceptions is not overhead when you need clustering of an open set of values. Unfortunately, the later provision of the standard-library exception hierarchy encouraged consistent use of full RTTI. This may be a violation, but that use wasn't anticipated (I was against the exception class hierarchy and my early examples passed and caught simple class objects whenever clustering wasn't needed). It is possible to optimize the handling of exceptions that are not clusters.

Exceptions may not be the best example of the zero-overhead principle, but they are not in obvious violation.

There has been little serious research on the topics of performance of exceptions and reliability of the resulting code in C++. There are, however, many small unscientific studies and lots of loudly expressed opinions – often claiming exceptions to be inherently slower than various

forms of checking of error-codes. That is not my experience. To the best of my knowledge, no half-way serious study has failed to observe that there are realistic examples where error codes win big and realistic examples where exceptions win big. In this context, “big” means integer factors, rather than a few percent.

Run a simple performance test: go N levels deep into a call sequence and then report an error. If the error is rare, say 1:1000 or 1:10000 and the call nesting is deep, say 100 or 1000, exception handling is much faster than explicit tests. If the call depth is 1 and the error happens 50% of the time, explicit tests win big. My naive, but potentially useful, question is “how rare must an error be to be considered exceptional?” Unfortunately, the answer is “that depends.” It depends on the complexity of the code, the hardware, the optimizer, the implementation of exception handling, and more. C++ exceptions were designed assuming an answer at least in the 1:100 region. In other words, that propagation of error indicators is far more common than explicit handling. The space problem is likely to be harder to solve than the run-time problem. For systems relying on termination, I could imagine an implementation simply terminating on a throw but if errors are to be propagated and handled, the tradeoff difficulties won’t go away.

Exceptions have been observed to speed up critical code by eliminating repeated test and by shortening the critical path. Exceptions have been successfully used in systems with low-latency constraints and real-time constraints. Not all exception performance stories are negative.

The N+1 problem

Whatever solution we come up with won’t make the old solutions go away. Even **errno** is still with us after 47 years. We’ll still have **errc**, the standard-library exception hierarchy, various error-code solutions (including **expected** and **outcome**), **pair** used to indicate error state, **nullptr**, and iterator pairs. These are not all standard, but those and more (including Microsoft’s “structured exceptions”) are widespread and actively promoted. Adding any solution will add one more technique to the language and/or library. It will add implementation complexity (if it is an ISO standard or in a platform standard) and users will have one more alternative to consider when using other people’s code and writing their own.

C++ exceptions came from such a simplification/unification effort when the world was far simpler and less diverse. Today, a unification of error-handling approaches seem even less likely to succeed.

Why don’t all people use exceptions?

Even people who theoretically could have used exceptions didn’t for a variety of reasons:

- Some people could not use exceptions because their code already was a large irreparable mess of unprincipled pointer use. Quite often, such people directed their critique toward exceptions rather than their old code.
- Some people (many) simply didn’t understand or even didn’t know of RAII and used exceptions as merely as an alternative return mechanism mirroring the use of error-return codes. Typically, code using try-catch as a form of if-then is uglier, larger, and slower than proper use of error codes or RAII.

- Some people started using exceptions by littering their code with **try**-blocks (e.g., inspired by Java). This leads to ugly, hard-to maintain code. For example, clean-up using **try { … } catch { … throw; }** can be 50-to-70 times slower than placing the clean-up code in a destructor.
- Many implementations of exceptions were slow because implementers generalized to handle other kinds of exceptions (e.g., Microsoft's "structured exceptions"), prioritized debugging (e.g., GCC walks the stack twice after a throw to preserve backtraces), used a single mechanism to serve a variety of languages (equally badly), or simply didn't expend much development effort on optimization.
- Exception handling have become relatively slower over the years because significant efforts have been spent optimizing non-exception cases. I suspect there are significant optimization opportunities left for C++ exception handling. For example, Gor Nishanov reported up to 1000 times speed improvements for some simple optimizations related to coroutine implementations on Windows and Linux [Nishanov]. Significant space improvements will likely be harder to achieve, though.
- Catching an exception is done by specifying the type of exception to be caught. As a result, the implementations of throw and catch got entangled with the mechanism for runtime type information (RTTI). This caused inefficiencies and complexity. In particular, it caused memory to be consumed (by the data needed for RTTI) even if an application never relied on RTTI for distinguishing exceptions and precluded optimization for simple cases. Furthermore, relying on RTTI made it hard to optimize the type matching where dynamic linking was used. Basically, exception handling implementations were tuned for the rare most complicated case. This was made worse when the exception class-hierarchy was added to the standard-library and people were encouraged to use that for even the simplest cases.
- The fact that exceptions are part of the platform ABIs makes it very hard to change early overdesigned implementations.
- Some people insist that only a single method of error handling be used and usually concludes that since exceptions are not suitable for every case, that method must be error-codes. The problems with error codes (e.g., forgetting to test and writing handlers) are then deemed "just inconveniences."
- Some people simply believed the persistent rumors of inefficiencies based on worst-case scenarios and/or unrealistic comparisons, such as leaving error-code handling in place after adding exceptions, comparing incomplete error-handling to exception-based handling, or using exceptions to handle ordinary choices rather than for handling errors that cannot be handled locally. There has been far too little serious investigation into the cost of exception and its alternatives. I suspect that the myths about exceptions have had more influence than any fact.

Most of these problems will persist with an N+1 solution (or more likely N+m solutions) as multiple competing error-handling proposals are being worked on.

"Deterministic exceptions" and error codes

"Deterministic exceptions" are a formalization of the use of error codes. The obvious implementation is turning a return value of type **X** into a pair **{error_code,X}** possibly optimized into an equivalent to **variant<error_code,X>**. The need for "formalization" comes partly from the need to optimize and partly to automate error propagation (up the call stack). I assume that every "deterministic exception" scheme will implicitly call destructors during propagation. Otherwise, RAII would be lost.

Pure error-code schemes require an explicit test for each function call, leading to the error-code hell known from languages such as Go. I see the main value of "deterministic exception" formalization as helping to avoid that.

Both "deterministic exceptions" (as proposed) and pure error-code schemes differ from C++ exceptions by being opt-in. I consider it impractical to change the massive amount of existing code depending on implicit propagation of exceptions. Also, many people (me included) consider implicit propagation to be the ideal.

At least part of the discussion of C++ exceptions and "deterministic exceptions" is based on a confusion between the definition of exception handling mechanisms and their implementation.

Costs

When considering error handling, three run-time costs must be considered

- *Error propagation costs*: the cost of getting an error indication from one scope into some enclosing scope where it can be handled.
- *Error recognition costs*: the cost of recognizing an error indication as relevant (to be handled, rather than to be propagated)
- *Error identification costs*: the cost of passing information related to the error along with the error indicator to a handler (e.g., a string or a list of problems to be addressed).

These costs typically involve both data and code, both time and space.

Run-time costs are not the only significant costs. We must also consider:

- Code complexity (probability of logic errors and lost optimization opportunities)
- Danger of errors not getting caught (cost of crashes)

Too often, discussions about exceptions have focused exclusively on performance and sometimes exclusively on performance of highly tuned code.

Exception handling

C++ exception handling is usually implemented using a table lookup based on the program counter to find exception handlers during stack rollback. Alternative schemes based on "markers" in stack frames (allowing an implementation to distinguish between normal return actions and exceptional returns) were tried early on, but their performance was deemed inferior and could be used only where C-language stack frames were handled compatibly.

I think that the current implementations of C++ exception handling are sub-optimal from a performance standpoint and could be improved:

- GCC always (even in optimized modes) walks the stack twice to provide better debug support.
- Implementations use complete RTTI implementations to do type matching (ostensibly to simplify use of dynamic linking).
- Cross-function optimizations of exception handling are still rare.
- Special-purpose stack-unwinding and type-matching algorithms are not used for systems with special requirements (e.g., with no dynamic linking or tight memories).
- Some exception handling mechanisms cater for generalizations and alternatives not mandated by the standard.
- There seem to be no optimization of the original case of passing and catching simple exceptions by value. Given **final**, we can know that a given exception type isn't the root of a hierarchy.
- Use of the general free store for all exception objects, rather than pre-allocated memory for common and important exceptions as anticipated in the original design.

One reason for the relative poverty of optimizations seems to be that since exceptions were considered slow there seemed no reason to optimize them. Thus, exception handling is now relatively slower than it was in the 1990s. Other kinds of code have been significantly optimized since then.

As an example of a missed opportunity, I can mention the Chinese-remainder fast constant-time type matching algorithm that I published in 2005 [Gibbs]. For closed systems and relatively small class hierarchies (as are increasingly common in embedded systems), this eliminates almost all of the type matching cost.

It may also be the case that the table-based implementation approach is sub-optimal for tiny memories compared to error-code/stack-marking implementation approaches. The possibility of special-purpose implementations of exception handling for special-purpose systems ought to be explored. The Edinburgh experiments [Renwick] are encouraging for this direction of exploration.

Unfortunately, significant improvements are likely to be ABI breaking, but that is likely to be less trouble than breaking the basic model of error handling.

Error-codes

Consider a "deterministic exception" throwing function

```
X f() throws { … return x; }

X void g() … { … X xx = f(); … return xx; }
```

Here, assuming the kind of implementation that has been most prominent in the discussions, a pair **{error_code,X}** must be returned from **f()**. That pair must be unpacked for **xx** to be accessed and either handled in **g()** or repacked and passed on as the result of **g()**. This packing and unpacking will complicate return-value optimizations (and could increase register pressure) and the testing could be costly. We need a good estimate of that cost in time and space.

If **f()** and **g()** are part of a long call chain from a caller that knows what to do about the exception to the point where the exception was detected, the packing, unpacking, and testing could be rather expensive ***even when no exception is thrown***. This was one reason all implementations of C++ exceptions eventually ended up being table-based after some early implementations had relied on marking stack

frames. We could imagine optimizations, but such inter-procedural optimization would equally apply to C++ exceptions.

"Deterministic exceptions" are likely to inject conditional execution paths into expressions where you might not expect them (and where the alternatives are out-of-line for C++ exceptions). For example, consider **a+b*c** for matrices where **+** and ***** might throw.

Unless the "deterministic exception" is immediately handled in **g()** after the return from **f()**, destructors must be executed on the exceptional return path. Consider

X void g() … { string s; … X x1 = f(); vector<Gadget> vg; X xx = f(); … return xx; }

This implies that instead of the table-lookup implementation of destructor calls for C++ exceptions, we get a rat's nest of conditional branches. The cost of destructor calls is a major part of the non-determinism of C++ exceptions that bothers hard-real time users. From this point of view, "deterministic exceptions" are ill named (as people could mistakenly believe that their cost would be easier to estimate than the cost of C++ exceptions). The tools that we miss for estimating the run-time cost of exception propagation will also be needed for "deterministic exceptions." Similarly, we need solid estimates of the cost of the code space for conditions compare to the data space consumed by tables.

We have generally assumed that a "deterministic exception" can be represented by a single word or even a single bit. In either case, additional information will often be needed to be passed along (e.g., a message string or a list of actions to-be-taken). This was the observation that led to throwing objects, rather than integer values.

We could pack many different values into a single word. This leads to the need to avoid value clashes (e.g., is exception 4731 unique in the system?). It also leads to off-stack storage of exception state, such as strings (as in the standard-library exception hierarchy) and linked list of traces. This can be costly and involve allocation. People would also have to choose between off-stack storage of large exception objects and copying such objects during exception propagation. This has happened to the error-code based error-handling mechanisms in Go (https://blog.golang.org/go1.13-errors).

The representation of clusters of exceptions, such as the standard **exception** hierarchy would either have to be non-general (not shared among different libraries) or added as a standard-library "bolt-on" on the side of the "deterministic exception" mechanism.

C++ exceptions and deterministic exceptions

Assume that we have some variant of the "deterministic exception" idea. The C++ exceptions won't go away, so an implementation will need to cope with both and combinations of both.

Consider a call

void f() throws { … g() … }

If **g()** may throw a C++ exception and **f()** throws a "deterministic exception" we could

1. Ban the call of **g()**. This makes "deterministic exceptions" viral in that every function called by a function throwing one must be converted from using C++ exceptions. I don't consider that viable.
2. Convert an exception thrown by **g()** into a "deterministic exception" to be returned by **f()**. This implies the cost of conversion code unless the compiler can prove that **g()** never throws. That might be expensive – adding to the cost of both "deterministic exceptions" and C++ exceptions.
3. Require the call of **g()** to be annotated in some way (say by a **try**) so that we know that we need to convert an exception. This places the burden of determining whether **g()** can throw on the programmer. If the programmer is responsible and wrong (fails to label a call that throws with **try**) we get some form of UB. That would be unacceptable. If the use of **try** is mandated by the type system, we are back in case [1] with added syntactic noise. If not, we are back in case [2].
4. Just propagate the exception as usual (ignoring the "deterministic exception" specification). This makes the "deterministic exception" specification untrustworthy. I don't consider that viable.

We have a similar case if **g()** can throw "deterministic exceptions" and **f()** can report errors by C++ exceptions. Again, we must carry an exception conversion cost.

We badly need solid estimates of such conversion costs.

Also, we need strategies for using C++ exceptions and deterministic exceptions in larger systems that use both (e.g., because they use two libraries that use different kinds of exception handling). We also need to consider systems that also use error codes and conversion strategies that might work at scale.

Consider a callback

```
X f(Callback g) … { X x = init; … x=g() … return x; }
```

We have three alternatives for error-reporting from **f()** and three from **g()**:

- C++ exceptions
- "deterministic exceptions"
- noexcept

The callback can be a function object (e.g., a lambda) or a pointer to function.

This is a bit of a puzzle

- If **g()** can return a "deterministic exception" we have to unpack and test before assigning to **x**.
- If **g()** can return through a C++ exception we automatically bypass the assignment to **x**

Unless we constrain the type of **g()**, we need to handle both cases. For a utility function, constraining the type of **g()** would mean the need to overload to handle all cases. In both cases, we need to percolate the exception appropriately, yielding 4 or 9 cases (depending whether you count **noexcept**), For a simple pointer to function, this implies overhead.

If, as seems to be assumed, "deterministic exceptions" are to be incorporated into the type system (as **noexcept** unfortunately was), it is not obvious how to relate the alternatives to each other. Consider:

- **int (*p)(int)** **//** *may throw C++ exception*
- **int (*)(int) throws**

- **int (*)(int) nothrow**

How do we write code using the alternatives if they are distinct. The same problem emerges with lambdas.

In considering the "deterministic exceptions" and their interactions with C++ exceptions, I fear "the mother of all ABI breakages."

Visibility

One of the reasons for introducing exceptions was to allow users to separate "ordinary code" from "error handling code." The fact that exception propagation paths are invisible was (and is by many, including me) seen as a feature. For both code hygiene and performance reasons, I intensely dislike having my code littered with checks for unlikely (exceptional) errors. Consider:

```
void user ()
{
        vector<string> v {" hello "};
        for (string s; cin >>s; )
                v. push_back (s);
        v[3] += " odd";
        auto ps = make_unique <Shape>( read_shape ( cin ));
        Smiley_face face {Point{0 ,0} ,20};
        // ...
}
```

This example is artificial, but stylistically not atypical. The **user()** function offers many opportunities for unlikely errors: memory exhaustion, read errors, range errors, construction failures (e.g., deep in the hierarchy of **Smiley_face**). In addition, the use of a **unique_ptr<Shape>** protects against a memory leak. If we used explicit error-codes instead of exceptions, we would need at least seven error-checks in this function, doubling the amount of source code, plus a few more checks inside the various constructors. Without RAII (and its integration with exceptions) the code would bloat further still. On average, more code implies more errors. This point is often underappreciated by people who argue from small examples. For such small examples, "just one test" doesn't matter much and is relatively hard to forget.

There is a school of thought that many of such tests can be eliminated through the use of preconditions: let the caller make sure that the input is correct and suffer undefined behavior if the user gets it wrong. For most cases, I consider that infeasible. Users make too many mistakes.

On the other hand, some errors should be expected and for those checks of some form of error code is preferred:

```
ifstream f {"Myfile"};
if (!f) {
        // ... deal with error ...
}
// ... use f ...
```

Here, the error-code is hidden inside the stream state for ease of use.

There is a special case that deserves attention as an optimization opportunity. Often, for generality, a library must throw to enable multi-level propagation, but in many cases a user knows that failure is likely and checks. For example:

```
try {
        p = new X[a_lot];
}
catch (bad_alloc) {
        p = new X[a_few];
}
```

And

```
try {
        X x;
        // … do something if x constructs correctly …
}
catch (Bad_X) {
        // … do something else …
}
```

The common case often can be (and today sometimes is) optimized to a simple condition. If programmers could rely on that, interfaces could be simplified by relying more on exceptions. For example, dual-interfaces approaches like the one used in **<filesystem>** might be unnecessary. I actually like the filesystem's dual interfaces because a user can often decide whether a call is likely to lead to an error, but not all interfaces have that property.

Memory exhaustion

It has been suggested that memory exhaustion as reported by **bad_alloc** should be considered separate, and not as an exception. That's tempting, because memory exhaustion is the ultimate unpredictable event, and in many applications vanishingly rare. However, there are many

- cases where a **new** fails (throwing an exception) but a program can manage without the as much memory as requested or the exact memory requested
- programs that may not fail (for any reason but hard hardware failure) so that exceptions are be turned into something like "log and system reinitialize."

A general-purpose library cannot know which is the case for a program in which it is used. This is the main reason that WG21 decided that **new** should throw.

What cannot be done is unconditional termination. These use cases were prominent when we decided on using **bad_alloc** exceptions. At a CppCon'19 session, a suggestion for unconditional termination after memory exhaustion was met with about 10 concrete counter examples from current systems.

Some examples of network failure are logically very similar to **new** failure; **bad_alloc** isn't unique. In particular, any library that may suffer rare (exceptional) errors has to deal with problems logically similar to failure of **new** (e.g., a function that depends on network access). It is reasonable to consider whether the handling of **new** failure and logically similar exception could be optimized (e.g., but using **final** exception classes).

Kernel Code

It is often asserted that exceptions cannot be used in kernel code (and similarly constrained code). The absence of exception use is then sometimes used as an argument for exceptions being unsuitable, dangerous, and/or slow. However, not using exceptions in a kernel is a historical/political decision (many kernels were started before C++ was a viable choice as a kernel implementation language), not a fundamental technical problem. C++ exceptions have been successfully used in kernel code, yielding benefits compared to the conventional error-code styles. [Gylfason].

Concerns

In addition, I have some general, rather than language-technical, concerns.

There are more to exceptions than performance

The current discussion of exceptions (error handling) tend to focus on performance. However, error handling is a prime software engineering topic. I recommend a greater focus

- on notation (e.g., use of operator overloading)
- on the role of constructors, destructors, and RAII
- on the effects on readability from littering the code with tests and "annotations"
- on the guarantees offered (e.g., the guarantee that every error reported is handled)
- on the complexity and cost of passing information about an error up the call chain

In particular, test cases used for performance measurements should not be restricted to relatively small fragments of high-performance code involving only built-in types.

Complexity and mission creep

A novel feature always looks simpler than an older one. This is often because the problems with the older feature are well known and have been addressed, often with less elegant "patches" to address compatibility and interoperability problems.

Typically, a "clean" novel facility increases significantly in complexity, implementation costs, and possibly even in overhead as it works its way through the standards process and the process of integrating it into the various tool chains.

I think there is good reason to believe that this will happen to "deterministic exceptions." The area of the language, the standard library, and the problems of use are so central and delicate that it is hard to imagine otherwise.

C/C++ compatibility

There is a long tradition of C and C++ being only "almost compatible" despite strenuous efforts by many people (including me) to eliminate as many incompatibilities as possible. However, the wish for

complete compatibility isn't completely universal. Even if it was, two committees with different memberships, working on different working papers and with different delivery schedules are unlikely to come up with completely compatible solution at first try. In particular, I worry that the C++ needs for compatibility with existing exception-handling mechanisms and for support of destructors could lead to novel incompatibilities, rather than the hoped-for increase in compatibility. Early shipping or accepting a C implementation that didn't address C++ problems could do long-term damage to C++.

Conclusion

We don't have a viable alternative to C++ exceptions. We do have opportunities for improved optimization and use. If those optimizations of the implementation of C++ exceptions are ABI breaking, they should still be considered in preference to language changes.

We are not sufficiently certain of the general approach of "deterministic exceptions" to decide on a direction. In the "deterministic exceptions" direction, we risk a bad case of "patching patches to patches to … ." Gradual extension based on experience is manageable only where we have a firm view of direction.

We don't have sufficiently good data for a serious discussion of real-world overheads.

Caveat

The burden of proof of value and the proof of minimal harm is on the proposers of something new. Demonstrating flaws in the status quo or in a defense of status quo is not sufficient and can be a mere distraction. A solution to "your local problem" (for your company or your industry) isn't necessarily a solution for the global C++ community.

Acknowledgements

Thanks to all who took part in the reflector discussion prompted by a draft of this paper. Some issues raised there were beyond the scope of this paper.

References

1. Herb Sutter: *Zero-overhead deterministic exceptions: Throwing values*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf.
2. Niall Douglas: *SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1028r2.pdf.
3. Gor Nishanov: C++ Exception Optimizations. An experiment. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1676r0.pdf.
4. James Renwick et al: *Low-Cost Deterministic C++ Exceptions for Embedded Systems.* https://www.research.ed.ac.uk/portal/files/78829292/low_cost_deterministic_C_exceptions_for_embedded_systems.pdf. Proc 28th International Conference on Compiler Construction (CC2019).
5. Michael Gibbs and Bjarne Stroustrup: *Fast Dynamic Casting*. http://www.stroustrup.com/fast_dynamic_casting.pdf. Software - Practice & Experience. Vol 35, Issue 12. 2005

6. Damien Neil and Jonathan Amsterdam: *Working with Errors in Go 1.13*. https://blog.golang.org/go1.13-errors. 17 October 2019.
7. Halldór Ísak Gylfason and Gísli Hjálmtýsson: *Exceptional Kernel – Using C++ exceptions in the Linux kernel.*