Document number: P0364R0
Date: 2016-05-23
Project: Programming Language C++, SG14, SG1, EWG
Authors: Michael Wong, Sunil Srivastava, Sean Middleditch, Patrice Roy
Emails: michael@codeplay.com, ...
Reply to: michael@codeplay.com

Report on Exception Handling Lite (Disappointment) from SG14

… or, How I learned to stop worrying and love the Exception Handling …

Introduction

One of the highest major topic in demand by SG14 (by games developers, embedded device programmers and to a lesser extend, some financial/trading communities) is to create a either an alternate error handling mechanism or a of spectrum of exception handling practices that covers from one extreme unwinding across function boundaries (as we have today), and some form of light-weight return-code like status values. The proposal is to standardize such an alternate error facility available in C++'s STL, and set it as guidance so that any other open source library can start following similar practice.

This practice has some precedence, as it is similar to the dual error handling system in Boost date/time, in the Filesystem TS and the Networking TS, where there is both exception handling and an error code facility, because it is appropriate in that domain.

This paper summarizes the discussions at three F2F meetings at CPPCON 2015, evening session at Kona, and GDC 2016 with no special analysis at this point yet of what is best. This is an effective snapshot of this investigation by SG14, to enable more discussion and to invite contributions. We have also been collaborating with Lawrence Crowl who authored status_value and the papers on disappointment.

Given that different domains want different forms of light-weight error handling, this paper primarily focus on the interest of Games Developers. A separate paper will address those of embedded programmers, and financial developers.

Games perspective on exception handling

Patrice Roy did a study using basic test cases and checked them against several compiler, including GCC, and MSVC 2015 for several scenarios. Code is available on the website [1]. His preliminary data shows that exceptions behave badly when not handled, that they consume time, even when they are not thrown. The data require more tests, especially in cases where no throws ever occur. There is plan to actually implement a game, such as Quake which has been open sourced with and without exception to fully comprehend the costs of exception in a complex simulation system.

While the performance of exception is important and one of the question people have asked to be answered (and it is under continued study and help and collaboration is appreciated), the other equally important aspect is the bifurcation created by exception-aware and non-aware code such that exceptions break code for non-exception users.

An example to add, for instance, are how simpler data structures (flat_* being an easy example) can only ever offer the basic exception guarantee without making other implementation trade-offs. Even then, just implementing the basic guarantee can imply adding a lot of complexity to an error handling path that you don't even necessarily know that you need since there's no explicit in-your-face indication that code flow can be violated arbitrarily by something as simple as x = move(y).

More complex, intricate, and highly-tuned data structures and algorithms can run a similar risk. Likewise, see the compromises that have been made in unordered_*, the compromises made in variant, the problems in providing so much as a concurrent_set<T>::try_get(T&) function, and so on.

The cost of exceptions is farther reaching than simply the code that handles throwing or catching them.

The opposite side of the argument is that code without exception is maybe more broken. For example, what do we do with ctor or other «special» operations that fail when there is no exception support. The flat_* container invariants are hard to maintain when move / copy ops throw, but we have not discussed (to my knowledge) how to handle failure / disappointment in the absence of exceptions. I think this is a necessary part of the picture. What would the recovery code be like? How would flat_* or other containers have to be written to handle such disappointments?

While terminating the program does the trick for games and some application domains, it is also not a general solution.
The opinion from all gamers is that exception is turned off by default in shipping code. Most of that has to do with the wish the game systems need to have no jitter from behavior such as exception unwinding, that can interfere with buttery smooth Hollywood-style graphics. Furthermore, since exception is very much a part of STL, this prevents STL from being used in games. Worst, it also prevents Open source libraries that follow C++ guidelines with exceptions to be use and drives down participation. So it has led to a very much rebuild everything attitude, including EASTL, or other replacement home-grown libraries. This is a shame considering that games is one of the top three industries that use C++. The other two are Finance and Banking.

There is never any question of removing exception handling in C++, just a wish to start the conversation between the games industry and the committee and look at standardizing a light-weight error handling path in addition to exception handling. In the meantime, games programmer simply turn off exceptions, and this has created 2 sets of code divide, one that is aware and handles exceptions and one that does not.

For now, we will focus a bit more on the cost from a games perspective. We can consider two essential questions:

a) Cost of exceptions, when they are thrown. This includes runtime cost of searching and stack unwinding.
b) Cost of exceptions in a program that never, or rarely, throws them. This includes space cost and optimization penalty.

The traditional rule of thumb has been that the exceptions should be thrown in truly exceptional situations; though it is difficult to pinpoint this any rule further: for example, should exceptions be thrown to manage user error handling in a compiler? Or perhaps only for more rare situations such as allocation-failure. In the opinion of one of the author (Sunil), no for the first but yes for the second.

Still, given this rule of thumb, and the way game programmers have been doing things (i.e., not using exceptions at all), I think it is the second item that is more important for SG14: The cost of exceptions in programs that never, or rarely, throw exceptions. Most existing game code fits in this second category.

The contributors of costs in this scenario (b) are:

· The space cost of tables needed for stack unwinding and exception type matching

· The space cost in code for cleanup actions, mostly destructors of local objects. This is the major contributor.

· The cost of missed optimizations (both space and time) because of exceptions. Exceptions effectively create extra branches in control flow graph which ultimately inhibit optimizations.

We must admit that we do not have hard data about any of these costs at this point, short of instrumenting something like Quake with and without exceptions. The only way to get that data will be to take a game that is currently using exceptions and rewrite it using something else. That is an expensive experiment. However, I did take a game that was not using exceptions and compiled it with exceptions enabled, to evaluate the space-cost. The numbers, as shown by the 'size' command are

	Text	Data	Text+data+bss
EH Enabled	12680680	1174896	14629340
EH-ignore	12650742	1149216	14573722
ratio	1.0023	1.022	1.0038

Note that this is a large piece of game code that has been written without using any exception handling constructs. It can be compiled with exceptions enabled or disabled. The overhead of exceptions in this case is primarily from cleanup code to run destructors on stack variables.
The space cost of exception tables is visible in the 'Data' column, about 2%, though it is much smaller percentage of the whole program.
The runtime penalty of enabling exceptions cannot be determined by this experiment, because the program never really throws an exception.

What can SG14 propose?

In three meetings: the SG14 meeting at CPPCon 2015, the SG14 Kona meeting evening session and the SG14 meeting in GDC 2016, we reviewed and classified the current proposals for making exception handling more palatable for Games developers.
These proposals are:

Brianstorm idea	Proposal

1. *Lighten the load from the library. What are the throw points and how can that be mitigated.*	
2. *throw = UB, catches are magically gone*	Allow but ignore try/catch blocks; convert throw to undefined behavior and catches are magically gone.
3. *thread local handler for cases where exceptions might be thrown (IM, kills it) error*	
4. *category, system_error, as used by filesystem and networking*	
5. *throw = quick_exit*	
6. *_NO_EXCEPTION macro definition*	The _NO_EXCEPTIONS macro. I'm concerned this feature would be used to further divide the C++ language into two tracts, but I can see its practical utility: A macro, called _NO_EXCEPTIONS, would be defined when the compiler is running with a flag that disables exceptions. Standardizing this macro allows developers to write portable code that has special behaviour when exceptions are disabled. For example, this macro could be used to remove the 'at' function of 'std::vector' when exceptions are disabled.
7. *make noexcept(true) the default*	This is a variation on -fno-exceptions. That option disables them entirely, even if you do need or want to use them in some places. A variant that could be recommended for implementors is to instead offer a flag that makes noexcept(true) the default for undecorated functions.

	In this case, anyone uses conditional noexcept (e.g. the standard library) will still work correctly with types and functions that can throw, so long as those functions are marked up appropriately. This shifts the burden from having to mark up many functions as non-throwing to marking up a small percentage of functions as throwing. That is, we mark up the (ahem) exceptional case, rather than having to mark up the common case. (for industries that typically disable exceptions, at least.) A useful addition from the standard here would be a reliable cross-vendor way to detect if this behavior was enabled. I believe this can be done in the library by testing a noexcept trait on an undecorated function/constructor. This item still divides code into standard C++ (noexcept off by default) and non-standard (noexcept true by default), but it makes it far easier for developers who cannot enable exceptions to interoperate with other code written for other domains. That is, it narrows the divide without eliminating it.
8. *function decoration to assert non-throwingness, optimising accordingly*	This idea is to mark a function with some kind of strawman_nowind keyword, allowing exceptions to be passed through without terminating but without any unwinding information/support. That is, it is an assumption that any exception thrown by a callee is leading to a fatal application failure, so cleaning up after the exception is not particularly important. A use case brought up for games was that we may want the game to remain running and interactive after an exceptional circumstance during development so that a tester or developer could inspect the simulated environment and observe any effects of the failure, which may be key to fast and accurate diagnostics. A variant of this may be recommendation for implementors to offer a -fno-unwind alternative to -fno-exceptions that always exhibits the above behavior; e.g. still allow throw and try/catch, but never generate unwinding information. This item still leaves a lot of code in a non-standard area, and any library or code that depends on exceptions for signaling errors will often result in incorrect behavior (as destructors may not be called), but it allows exception-adverse developers to interoprate with well-behaving libraries. That is, game developers could still use and contribute to good C++ libraries and not be forced to fork or rewrite nearly as much code; throw and try/catch would still compile without error, and problems would only arise if an actual erroneous situation (for well-behaving libraries that don't use

	exceptions for code flow) occurs.
9. *standardising industry practice e.g. —noexception*	The simplest choice is just allow no-except, since every compiler has this as an option. 1) Just standardize the current practices with NO changes to the language or the library. Under the principle, that ‘the purpose of the standard is to standardize the current programming practices’, we should do at least this much. Every compiler implements options of disabling exceptions, and most game programmers use them, though in different ways. I realize that the language standard does not address issues of compiler options, and does not even acknowledge different ‘modes’ of execution, but the facts on the ground are that no-exceptions modes are being widely used.
10. *tagging a class or namespace or module as noexcept*	This is a language-level extension that serves to make it more palatable for developers to leave exceptions in their default enabled state. With exceptions enabled, developers that operate in environments adverse to exceptions must remember to mark up even single function they declare as noexcept or conditionally-noexcept. An option is to allow a developer to tag a namespace, class, or module as noexcept, which would set the default noexcept mode for just the enclosed declarations. e.g., instead of marking every function on a class as noexcept, the developer just marks the class as noexcept. Likewise, if a developer is writing a new file or module that is going to contain only noexcept functions/classes, it would be convenient to indicate such with fewer repetitive keywords. This item potentially can completely remove the divide between standard C++ and exception-free environments. This at first seems to be "just" a convenience feature, but one that makes it far more palatable for developers who might otherwise rely on -fno-exceptions to leave them in enabled with fully standard behavior.

11. *error or exceptions through expected or coroutines*	struct error_t {};
	constexpr error_t error;
	
	template <typename T>

```
struct expected {
    int err;
    T val;

    expected() {}
    expected(T value) : err(0),val(value) {}
    expected(error_t, int err) : err(err) {}

    bool has_value() { return err == 0; }
    bool has_error() { return !has_value(); }

    T const& value() const { assert(has_value()); return val; }
    T & value() { assert(has_value()); return val; }

    int error() { assert(has_error()); return err; }
};

expected<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337, block);
    for (;;)
    {
        auto bytesRead = await conn.read(buf, sizeof(buf), block);
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

Other languages

We believe there is value in studying the recent improvements from Swift and Rust. This is not a "omg let's do exactly what Swift does" email (in fact, please don't) but rather just something we think could be informative to those who wish to attempt to wrestle the "exception problem" in C++ in some fashion or another. If you don't already agree that there _is_ an exception problem in C++, then this will not change your mind.

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html

(a simlar model can be found in http://joeduffyblog.com/2016/02/07/the-error-model/)

A rambling summary and my thoughts follows.

First, Swift does have an actual exception model rather than the more function-style "result type". There's a real try keyword and a catch keyword, the ability to throw from a function, and so on. In this way, the model is closer to C++ than, say, Rust.

The differences start stacking up relatively quickly. Possibly the biggest - and the one that fits my personal tastes really closely - is that exception propagation in Swift is explicit. That is, to call a function that might throw, you must explicitly say:

```
try some_fallible_function(); // propagates the exception
```

or if it has a return value:

```
let x = try some_fallible_function(); // propagates the exception
```

This has two big consequences. First, from a taste perspective, it has an explicit model. It becomes very very obvious in a function when an exception could interrupt control flow, and so it is much easier to verify that a function will not have postconditions broken by an unexpected "return" to the caller. Second, it tells the optimizer much the same thing - it doesn't have to assume that every single function call might throw and hence can generate more optimal code with less waste emitted for unwinding purposes.

Swift more formally notes that in its exception model, a function that throws only propagates the exception into the "enclosing scope" i.e. the caller. Unlike in C++, a single throw cannot implicitly bubble all the up a callstack without the intervening callers acknowledging the possibility.

Note that it would be possible to offer a merger of this behavior and C++'s existing behavior for functions that can truly be exception-agnostic. Essentially, it would just place the burden on users to annotate a function that is exception-agnostic rather than having to annotate all the functions that never throw, e.g.

```
// explicit-per-statement model, as in Swift
T something() throws {
  foo(); // compiler error if foo() is not a no-throw function
  try bar(); // propagate exceptions
  return gaz();
}

// implicit model, closer to C++ noexcept(false)
T something() try {
    foo(); // compiler knows that foo() won't throw, but no errors if foo is marked as throwing in the
future
    bar(); // propagate exceptions, no 'try' required
    return gaz();
  }

  // no-throw by default
  T something() {
    foo(); // compiler error if foo() is not a no-throw function
    try bar(); // compiler error to not catch the error in the body of the no-throw function
    return gaz();
  }
```

Of course, adopting the explicit call nature in C++ directly is today a non-starter. Pretty much any function in C++ can throw and _might_ throw, largely because C++ still tries to try general memory allocation failure as a recoverable error. Swift (and a good number of other languages I've surveyed) assume that domains where memory allocation failure is likely and needs to be recoverable are using specialized containers and interfaces.

Making that same assumption in C++ (after a length round of deprecations and transition) could have wildly useful impacts on the language. Take the whole empty variant kerfuffle - we only need to support empty variants because move operations are very likely to be noexcept(false) in C++ today, and the vast majority of reasons there are because of potential memory allocation failures. If we assumed that the vast majority of types have non-throwing move semantics even in the face of allocations, a lot of "tricky" cases in C++ become rather trivial, and we don't really lose a lot (personally, I can count the number of times I've seen someone catch std::bad_alloc on the fingers of a bowling ball...).

Back on Swift, because so few functions in Swift throw, they have a different default than C++. In Swift, functions are no-throw by default.

Swift has two variants of the try keyword. The first variant is "abort on exception":

```
  let x = try! some_fallible_function(); // I promise this won't fail... and if it does, kill me now
```

This is to allow a non-throwing function or a function with no sensible recovery strategy to call a throwing function that is unlikely to fail. e.g., in a game, I might not care about I/O errors opening a core resource distributed with the game. If the user went in and deleted required files, then the game is going to crash and nobody in the world should be surprised.

The second variant is arguably much more useful, though I personally feel that Swift made a slight error. This variant is the "wrap in an optional" choice:

```
 let x = try? some_fallible_function(); // returns an optional, no propagation
 if x {
   do_something_with(x);
 }
```

A problem with this exact implementation is that the error itself is lost, which is ungood. This variant is only useful if an error is recoverable but all errors are recoverable in the same way... _and_ if you have no reason to log diagnostic information. A much more useful approach IMO would have been to return a Result type instead of an Optional type so that the error can still be extracted.

Note that the try keyword affects the entire following expression. This means that chaining calls it not an issue - if any part of the expression throws an exception, the remainder of the expression is unevaluated.

Lastly worth noting, in Swift all throwable types must adhere to a particular "protocol" (analagous to a concept in C++... thought Swift also supports "virtual concepts" semantics with its protocols... so they're also comparable to a Rust "trait"). This means that exception-wrapping and exhaustive matching is much easier, as Swift doesn't have to deal with weirdos trying to throw an integer or string literal. The language can detect if a try-catch block will catch all possible exceptions by seeing if the final catch block matches the base exception type.

The interesting aspect of Swift's approach over that of Rust or other functional languages are numerous.

First, ignoring errors must be explicit. This is an oft-cited weakness of the return value approach to error-handling that also affects Result type values. In Swift's model, you must explicitly use an operation like try! or try? to consume an error and handle it. Further, the try! variant doesn't actually _ignore_ the error, it just makes it a runtime error. A user that explicitly converts the error into an Optional/Result could ignore the error, but that's not really any different than a use of C++ today just adding an empty catch(...) handler.

Second, exception-based code is easier to adopt to Swift's model. Adding the try keywords to places where exceptions are expected is mostly a mechanical transformation. Again, in C++, this would be quite nasty unless the language _also_ vastly reduced the number of places that an exception could occur (by being noexcept by default, and not treating general allocation failure as a recoverable error). Convering entirely to a Result-based model would incur that same amount of work while additionally requiring a lot of function signature changes, though, so comparatively speaking Swift's model is still more palatable.

Third, Swift doesn't implicitly enforce exception type declarations. e.g., Swift functions that can throw just mark themselves with the `throws` keyword, while typical Result-based functions also include some kind of error type constraint. That is, `Result<T, IOError> read_file()` can only return errors of type IOError, which constrain future implementation choices while a function of the form `T read_file() throws` can return any potential error. This of course reduces the explicitness a bit as callers have to somehow be ready to handle or propagate any arbitrary error, but I personally feel that's a smaller issue than just knowing if you have to handle _any_ error or not.

Fourth and finally, Swift is built to be a little more compatible with C++ than most functional languages (Rust included) are. Swift, for all its newness and differences, is meant to easily interoperate with existing Objective-C and Objective-C++ codebases. Swift's design and implementation have to deal with a lot of the same kinds of back-compat problems that any C++ evolution would require.

Overall, _any_ change to C++'s exception semantics are pretty limited. Simply adopting the Swift model is outright impossible (and not necessarily desirable) without breaking an awful lot of code. That said, there's probably something to learn here.

A digression to see what Embedded group want:

This group would like "exceptions handling lite", which has 3 versions:

1) exceptionless exception handling ("EEH"), which could be renamed as just unwinding. This has the benefit of allowing RAII and no allocation of the exception object (indeed, since an exception can be thrown due to lack of memory, library implementations usually reserve an extra buffer for this, usually called EMERGENCY_BUFFER), plus avoiding the overhead of the exception landing procedure (including the costly reflection to traverse the inheritance graph to check the catch clause). Syntax is just throw() and catch(). We would need [p0132r0] plus a few modifications to nothrow new, specifically a nothrow_new_handler and not require nothrow new to "retry" in a loop when no memory, but just call the handler and return nullptr.

2) exact match catch exception handling ("EMCEH"). This has two flavours, but basically the idea is to avoid the inheritance traversing, so the catch succeeds only if the thrown type is exactly the caught type. The prototype has no syntax, it's just a hacked gcc. The traversing takes more time when the inheritance graph involves virtual inheritance. Initial reports of a prototype showed that, at least in IA32, the traversing time of the inheritance graph of less than 10 nodes is negligible.

3) statically determined catch landing site. We did not proceed with this option so far, but requires whole program analysis in order to determine where a throw will land at compile time. Requires more elaboration and is still sketchy.

Conclusion

None as yet.

Acknowledgement

I wish to thank SG14 as much of this is a snapshot of the SG14 F2F discusssions and threads from many authors.

References

[1] *http://h-deb.clg.qc.ca/Sujets/Developpement/Exceptions-Costs.html*
[p0132r0] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0132r0.html
[P0262R0] A class for STatus and Optional Value,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0262r0.html
[P0157R0] Handling Disappointment in C++;
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html