P1095R0/N2289: Zero overhead deterministic failure – A unified mechanism for C and C++

Document #:	WG14 N2289
	WG21 P1095R0
Date:	2018-08-29
Project:	WG14 Programming Language C
	WG21 Programming Language C++
	WG21 Evolution Working Group
	WG21 SG14 Low Latency study group
Reply-to:	Niall Douglas
	<s_sourceforge@nedprod.com>

A proposed universal mechanism for enabling C speaking programming languages to tell C code, and potentially one another, about failure and disappointment.

This proposal is being concurrently submitted to both WG14 (C) and WG21 (C++), as it is proposed that C++'s proposed deterministic exception throws [P0709] be implemented using a compatible superset of this universal C mechanism.

Contents

1 Introduction

This is an unusual paper, being submitted concurrently to both to WG14 (C programming language) as well as WG21 (C++ programming language). It originated from WG21 SG14 the Low Latency study group, where most of our group would tend to work more frequently with C-ish code than other WG21 members.

C++ is considering making its exception handling implementation value-based instead of type-based in order to make it deterministic, see [P0709] *Zero-overhead deterministic exceptions: Throwing values*. We could come up with some local-to-C++ solution, but I think we can do a lot better by closing the gap between the C++ ABI and the C ABI such that C code will be able to legally call, *without intermediate bindings*, C++ functions capable of throwing value-based exceptions (so long as such functions do not receive nor return C-incompatible types). C code would directly understand that a C++ function it called had failed – moreover, C code would be able to return exception throws to C++ code.

If this coordination can be pulled off, the benefits could be profound for all C speaking programming languages, for example Rust, Python or Fortran. All these, being able to speak C, could directly understand and generate C++ exceptions.

Outside of C++, there are substantial benefits for C as well, because we can extend the proposed universal mechanism to allow individual functions to implement POSIX `errno` more optimally. Specifically:

- The use of POSIX `errno` to indicate cause of failure introduces a lot of unhelpful side effects both in C and C++, because it prevents some C standard library functions (the `<tgmath.h>` functions especially) being marked as pure, which in turn inhibits more aggressive optimisation.
- The use of POSIX `errno` introduces a hard dependency on thread local storage, which in turn forces million thread compute resources to not implement standard C for their compilers, as transporting thread local storage to each CPU is impractical.

And finally, this proposed mechanism makes it much easier for C to import Contracts from C++, as a number of people on the WG14 reflector appear keen to do, as the contract violation handler can do something other than abort the program, which is very useful for unit test code ensuring that contract violation is detected.

2 Proposed Design: C

It is proposed that C add the following:

- A new function declaration and definition attribute `fails(E)`, where `E` is a type. This modifies the type of the function to indicate that it has a different calling convention to ordinary C functions, specifically that now two channels of function return are possible, *successful* and *failure*. Successful is the ordinary return type of the function (if any). Failure is the type `E`.

- A `failure(expr)` used to tell the compiler to return the evaluation of the expression via the failure return channel, rather than via the success return channel.

- A `catch(function(...))` which calls the specified `fails(E)` marked function, emitting a designated aggregate initialiser of a type matching the form `struct { union { T value; E error; }; _Bool failed; };` for a function of the form `fails(E) T funct(...)`.

  To clarify what I mean here, if the function failed, `catch()` emits an aggregate initialiser with the designates `{ .error = returned failed(expr), .failed = 1 }` which is suitable for initialising any aggregate type with members of `error` and `failed`. If the function succeeded, `catch()` emits an aggregate initialiser with the designates `{ .value = returned value, .failed = 0 }`. It is up to the programmer to define some type suitably matching this designated aggregate initialiser.

- A `try(function(...))` is a convenience piece of boilerplate equivalent to:

```
struct { union { T value; E error; }; _Bool failed; } __unique_temporary = catch(function(...));
if(1 == __unique_temporary.failed)
{
  return failure(__unique_temporary.error);
}
```

  Writing the above boilerplate is very common in code with success and failure return channels, and saving having to write it out by hand every time is user friendly.

It should be noted that all of the above are *convenience macros* to new underscoreCapital keywords, as follows:

```
// Proposed <stdfails.h> header

#define fails(E)      _Fails(E)
#define failure(expr) _Failure(expr)

#ifndef __cpluscplus

#define catch(f)      _Catch(f)       // catch extended at C++ language level
#define try(f)        _Try(f)         // try extended at C++ language level

#endif
```

Some may feel that the collision potential of these choices of naming is too high, however it is the same mechanism which C99 `<stdbool.h>` uses, which macro defines `bool` to `_Bool` if not in C++.

One might wonder why not return `struct { union { T value; E error; }; _Bool failed; };` directly from the function? One can do this of course, however the `fails(E)` approach has a number of advantages:

1. The C calling convention has the caller allocate the space for the returned value before calling a function. `union { T value; E error; }` takes the size of whichever the bigger of types `T` or `E` is, which is as optimal as it can be. However the additional *discriminant* in `struct { union { T value; E error; }; _Bool failed; };` could take up to eight additional bytes more than that, or worse, depending on packing. This may not seem like much, but it is more than optimal.

   For `fails(E)` returns, it is proposed for at least AArch64, ARM, x86 and x64, that the discriminant be returned via the CPU's carry flag. This is because compilers can often fold the setting or clearing of the CPU's carry flag into the ordering of other operations, thus making this a zero runtime overhead choice of discriminant[1]. On other architectures such as RISC-V (which has no status register), an extra register would make more sense. It doesn't matter what an architecture chooses, so long as it is *consistent* across all compilers.

2. Calling a `fails(E)` function without wrapping it in either `try()` or `catch()` would be a compile time error. This requires the caller to *explicitly specify* how to handle failure. We could not be so enforcing with returns of the `catch()` output type as we cannot know what the programmer meant.

An example of the proposed syntax:

```c
int some_function(int x) fails(float)
{
  // Return failure if x is zero
  if(x != 0)
    return 5;
  else
    return failure(2.0f);
}

fails(float) const char *some_other_function(int x)
{
  // If calling some_function() fails, return its failure immediately
  // as if by return failure(some_function(x).error)
  int v = try(some_function(x));

  return (v == 5) ? "Yes" : "No";
}

#define caught(T, E) struct caught_ ## T ## _ ## E { union { T value; E error; }; _Bool failed; }

int main(int argc, char *argv[])
{
  if(argc < 2)
    abort();

  caught(const char *, float) v = catch(some_other_function(atoi(argv[1])));

```

[1]As it is a single bit being branched upon, status register update pipeline stalls should not occur.

```
  if(!v.failed)
  {
    printf("v is a successful %s\n", v.value);
  }
  else
  {
    printf("v is failure %f\n", v.error);
  }
  return 0;
}
```

And running this program would yield:

```
ned@lyta:~/windocs/boostish/wg21$ ./test 0
v is a failure 2.000000
ned@lyta:~/windocs/boostish/wg21$ ./test 1
v is a successful Yes
```

2.1 Extension: Purifying `errno` setting functions

A long time ago POSIX chose to return the cause of failure of a function by setting a then-global variable `errno`. This has since become a thread-local variable, but the side effects of calling functions which modify `errno` impedes optimisation, and makes impossible the use of standard C in very large CPU core machines.

This particularly affects the C math functions. Let us take a simple example, making an integer positive:

```
int myabs(int x)
{
  if(x == INT_MIN)
  {
    errno = ERANGE;
    return INT_MIN;
  }
  return (x < 0) ? -x : x;
}
```

This sets `errno` if it fails with the cause of failure, which prevents us from marking this functions with `__attribute__ ((pure))`, or even `__attribute__ ((const))`. This means that the compiler must assume that calling these functions has side effects on global state. That in turn greatly limits the ability of the optimiser to perform common subexpression elimination and loop optimization on code containing math functions, which is unfortunate.

I propose to fix this via an extension of `fails(E)`, the `fails_errno` modifier, which is expanded by the compiler into boilerplate. So given this function:

```
int myabs(int x) fails_errno __attribute__((const))
{
  if(x == INT_MIN)
  {
    errno = ERANGE;
    return INT_MIN;
  }
  return (x < 0) ? -x : x;
}
```

This is transformed by the compiler into as if:

```
inline int _Catchable_myabs(int x) fails(struct { int /*errno*/, int /*failure return value*/})
    __attribute__((const))
{
  if(x == INT_MIN)
  {
    // Do not set errno, return what it would be set to
    return failure({ERANGE /*errno*/, INT_MIN /*failure return value*/});
  }
  return (x < 0) ? -x : x;
}

int myabs(int x)
{
  caught(int, struct { int /*errno*/, int /*failure return value*/}) r = catch(_Catchable_myabs(x));
  if(r.failed)
  {
    errno = r.error.errno;
    return r.error.value;
  }
  return r.value;
}
```

I should stress the 'as-if' nature of the above. In practice, `myabs()` would be generated with a preamble and epilogue which sets real `errno`. If, and only if, `myabs()` is called by 'new code' (see below) would the setting of real `errno` be avoided by calling the function in a way which skips that preamble and epilogue e.g. at some fixed offset from the symbol, or by supplying a different return address etc. This allows existing backwards binary compatibility to be maintained such that unrecompiled code can call newly compiled code, but allows code compiled with new compilers to delay setting real `errno`.

2.1.1 Delay setting real `errno` in newly compiled code

The more useful part of this extension is what happens when we call a function marked `fails_errno`.

If in newly compiled code the compiler calls a function marked `fails_errno`, some more mechanistic transformation is performed to delay the setting of real `errno` to as late as possible. Consider the following code typical of any `errno` based failure detection:

```
errno = 0;         // clear errno
int v = myabs(x);  // call potentially failing function
if(errno != 0)     // if errno non-zero, function failed
{
  fprintf(stderr, "abs(x) failed with code %d\n", errno);
}
```

Transformed code would be as if:

```c
caught(int, {int /*errno*/, int /*failure return value*/}) r = _Catch(_Catchable_myabs(x));
int v;
if(!r.failed)
{
  v = r.value;
}
else
{
  // As we don't know if fprintf() reads errno, cannot set errno
  // any later than this unfortunately.
  errno = r.error.__errno;
  fprintf(stderr, "myabs(x) failed with code %d\n", errno);
}
```

But if `fprintf()` were also marked `fails_errno` ...

```c
extern int fprintf(FILE *, const char *, ...) fails_errno;
...
caught(int, {int /*errno*/, int /*failure return value*/}) r1 = _Catch(_Catchable_myabs(x));
int v;
if(!r1.failure)
{
  v = r1.value;
}
else
{
  caught(int, {int /*errno*/, int /*failure return value*/}) r2 = fprintf(stderr, "abs(x) failed with
      code %d\n", r1.error.errno);
  // Original code did not test for success/failure of fprintf,
  // so do nothing here for now
}
...
// Somewhere later in the function, set errno as late as
// is possible from the last executed fails_errno function
// in order to not violate current errno semantics
errno = r2.error.errno;
```

In other words, we are hoisting the setting of real `errno` out of the callee and into the caller, which in turn may get hoisted even further up the call stack, ideally if possible to the `main()` function such that real `errno` almost never gets read nor written at all.

One may have noticed that `fails_errno` functions have no means of reading `errno`. This is intentional: functions which need to read `errno` could not have the `fails_errno` attribute applied to them. In order to help trap reads of `errno` when porting existing code to `fails_errno`, I would thus propose that the following be a compile error:

```c
int func(...) fails_errno
{
  // ERROR: "Illegal to read errno before writing it in a _FailsErrno function"
  if(errno == 0) ...
}
```

2.1.2 How to completely eliminate setting of real errno?

So far, we simply delay the setting of real errno by hoisting it as high as is possible out of the execution graph. Some people will want real errno to never be set at all, thus guaranteeing purity of functions, and thus maximum common subexpression elimination.

My proposal is that for functions marked fails_errno_invariant, real errno modification is guaranteed elided by the compiler:

```
int func(...) fails_errno_invariant
{
  ...
  x = myabs(y);
  if(errno != 0)  // errno not actually modified, as per transformation above
  {
    ...
  }
  ...
  // Safe to call extern errno setting functions if errno is saved
  // and restored like this
  int olderrno = errno;
  fprintf(stderr, "Hi!\n");
  errno = olderrno;
  ...
  // No lazy setting of real errno performed
}
```

All fails_errno_invariant does is to cause the compiler to skip lazily setting errno in any hoist. It is on the programmer to not do anything which causes errno to become modified on function exit, as the compiler can hard assume that calling func(...) above will never modify errno.

3 Proposed Design: C++

This section is necessarily quite involved, as the C mechanism just proposed is calling convention compatible with how I propose C++ implements [P0709] *Zero-overhead deterministic exceptions: Throwing values*.

Given the WG14 audience, I will go into more detail than I normally would, for a purely WG21 targeted paper, regarding how C++ 20 currently works for those not familiar with modern C++. Those not interested in C++ particulars can feel free to skip this section, though I have started from first principles in order to ensure that no knowledge of the C++ standard is required to understand this section.

3.0.1 Background: C-compatible types in C++

Before the C++ 20 standard retired the trait, there was a concept that C-compatible types in C++ matched a concept called Plain Old Data (POD). C++ 11 added a trait std::is_pod<T>, but 'POD types' were well established into the C++ programmer's vernacular long before C++ 11.

In C++ 11 parlance, POD types are those which are (i) trivial (`std::is_trivial<T>`) and (ii) have standard layout (`std::is_standard_layout<T>`).

Trivial types, in turn, are those which are (i) trivially copyable (`std::is_trivially_copyable<T>`) and (ii) have a trivial default constructor (`std::is_trivially_default_constructible<T>`).

These subcategories of C-compatible types are important to understand, because while you may not be able to construct many C++ types in C, you are able to legally pass a larger subset of C++-only types *through* C. Such C++ types may thus be transported by C code via the proposed C `_Fails(T)`.

There are two specific subcategories of C++ types which C can transport safely:

1. **Trivially copyable**
    - Every copy constructor is trivial or deleted.
    - Every move constructor is trivial or deleted.
    - Every copy assignment operator is trivial or deleted.
    - Every move assignment operator is trivial or deleted.
    - At least one copy constructor, move constructor, copy assignment operator, or move assignment operator is non-deleted.
    - Trivial non-deleted destructor.

    As trivially copyable types must have a trivial destructor, C can copy such types without concern, as destruction has no side effects.

2. **Move relocating** (if [P1029] *SG14 [[move_relocates]]*, or an equivalent, is accepted)
    - Copies can be non-trivial.
    - Moves can be non-trivial, so long as moves are equal in effect to a memory copy of bytes from source to destination, followed by memory copy of default constructed instance to source.
    - Destructor can be non-trivial, but must have no side effects when executed on a default constructed instance.

    As move relocating types have a destructor which is guaranteed by the programmer to have no side effects when called on a default constructed instance, and that a moved-from instance has the same representation as a default constructed instance, C can copy such types without concern, even though it will not overwrite the source with a default constructed instance. This is because C would never call C++ destructors in any case, so it does not matter if the source is simply discarded.

    The caveat here is that while move relocating types may pass *through* C code, they must always return to C++ code before their life ends, otherwise it is undefined behaviour.

3.0.2 Background: How C++ currently implements exceptions

C++ exception throws take the form of a `throw expr`. This causes the stack to be unwound until a `try` wrapped statement block has a `catch(type)` matching the type of the value thrown. Types are matched using a Run Time Type Information (RTTI) search, making them non-deterministic as the RTTI in a process is unknowable in advance (and can change over time in a given running process as shared libraries are loaded and unloaded).

Back in the 1990s, C++ exceptions were implemented by setting an unwind handler on entry to each stack frame. When one threw an exception, the handler for each stack frame on the stack would be executed in order to unwind the stack.

This setting up of unwind handlers per stack frame executed added a fair bit of overhead on the CPUs of the late 1990s and early 2000s, so stack frame based handling was generally replaced with table based handling instead, which has remained until now. At compile time, the compiler generates a set of lookup tables for all C++ code compiled. When an exception is thrown, a runtime library routine is called which uses the current instruction pointer and the stack frames on the stack to look up in the EH tables which unwind handlers to invoke. This enables the successful code path to execute with effectively zero runtime overhead, but at the cost of bloating the executable binary with EH tables, and making the failure code path execution time highly non-deterministic.

However, CPU technology has marched onwards. The cost relative to CPU speed of scanning EH tables during a stack unwind has risen exponentially over what it was in the early 2000s, as such tables are inevitably in cold memory out of cache (and sometimes even paged out, on disc), and thus each table entry access costs at best hundreds of CPU cycles. One naïve benchmark puts the average *lower* bound cost at approximately 2,000 CPU cycles per stack frame unwound[2]. Upper bounds tend into the hundreds of milliseconds on a machine with limited free RAM, due to page faulting in the EH tables.

C++ has also moved closer to the hardware within the technology stack since the early 2000s. It is not commonly used as a general purpose application programming language in new projects where indeterminacy is acceptable; for new projects it is increasingly going into use cases where determinacy for both success and failure is paramount (reliability critical systems such as car self driving), and where the executable bloat generated by the EH tables is considered unacceptable (the embedded domain is the obvious one).

Finally, modern CPUs now tend to speculatively execute out-of-order rather than the in-order mechanism of earlier CPUs. Adding explicit checks for whether an exception has been thrown to the successful code path is now usually free of cost on such CPUs, as stalls in other parts of the pipeline will block the CPU, thus leaving idle spare CPU execution units. As much as it might be tempting to return to the stack frame based handling approach, modern CPUs prefer potential jumps to known branches, rather than potential jumps to unknown installed unwind handlers, as the former suit their branch predictors better. Thus was born [P0709] *Zero-overhead deterministic exceptions: Throwing values*, which recognises that the hardware has changed, and the typical use cases for C++ in new code have changed.

[2]Source: `https://ned14.github.io/outcome/faq/#what-kind-of-performance-benefits-will-using-outcome-in-my-code-bring`

In P0709, lightweight exception throwing functions are separated from legacy exception throwing functions via a `throws` modifier. Here are some examples:

```cpp
// May throw legacy type based exceptions
extern int func(int);

// Never throws any exception (noexcept was added in C++ 11)
extern int func(int) noexcept;

// May throw new value based exceptions, proposed by P0709
extern int func(int) throws;
```

3.0.3 Background: `std::expected<T, E>`

[P0323] *`std::expected`* is in the final stages of standardisation, and it proposes a standard A-or-B sum type for C++. This vocabulary type provides:

- An observer (`.has_error()`) to check whether it contains a value (`T`) or an error (`E`).
- An observer to access the value, if present (`.value()`).
- An observer to access the error, if present (`.error()`).

[P0786] *ValuedOrError and ValueOrNone types* proposes a C++ Concept `ValueOrError` which matches all types providing at least the observer interface listed above. The expectation is that types able to consume `ValueOrError` concept matching types would provide constructors which do so. One such type, obviously enough, would be `std::expected<T, E>` itself which would construct from any `ValueOrError<T, E>` concept match.

3.0.4 Background: The current proposed mechanism for deterministic C++ exceptions

It is legal in the current type throwing system to throw a non-copyable, non-movable type, and during unwind to throw lots more of them, which forces compiler implementers into some amazing gymnastics in order to meet the requirements of the C++ standard that this works correctly. Compiler implementers therefore must use extremely conservative implementations which are non-deterministic (e.g. dynamically allocating memory), in order to cope with what is a very rare, and barely used, use case.

One of the key proposals of P0709 is that the new exception throw mechanism throw *values*, not types, and that the only type throwable under the new lightweight mechanism would be a `std::error` which is defined to be no more than two CPU registers in size, and either move relocating or trivially copyable. This guarantees to the compiler that thrown values can be hoisted up the stack frame during an unwind exclusively using CPU registers, something which enormously simplifies implementation for compiler vendors, and which ought to lend itself well to aggressive optimisation.

P0709 in its current form could be read as meaning that `std::error` is a *lightweight token* to opaque, possibly reference counted or thread locally stored, state containing the ‘real’ exception state. If a token, it would permit `std::uncaught_exceptions()` to work as at present, and for throws of type based exceptions to use the value based unwind mechanism, rather than EH tables.

This would achieve most of the stated aims of the P0709 paper (elimination of EH table bloat, deterministic exception throws most of the time), and has obvious attractions for source code backwards compatibility – one is effectively throwing around `std::exception_ptr` instances which are well understood, and existing code is already using (see below for more detail).

However, in my opinion, a token-based deterministic exceptions implementation is insufficiently zero overhead or deterministic to warrant the title of the proposal paper. I therefore give additional background on SG14 low latency study group's alternative, hard deterministic, proposed `std::error` design.

3.0.5 Background: The alternative proposed mechanism for deterministic C++ exceptions

P0709 does suggest an alternative mechanism to throwing tokens to opaque exception state, and that is to throw the whole exception state in the value with no linked opaque data at all. Under this scheme, `std::error` is merely the *default* exception state type consisting of two CPU register sized members (the first is a pointer to a constexpr explanatory *domain* which inlines itself readily, the second is a code with meaning to that domain). If one annotates a function with merely `throws`, then that function deterministically throws `std::error` and nothing else. If one however annotates a function with `throws(T)`, then that function deterministically throws `T` instead.

This may seem like reintroduction of the hated dynamic exception specifiers, which were only just recently excised from the language [N3051]. These caused the program to terminate if the throw of an exception reached an exception specification boundary where the type of that throw was not listed. As one of the main use cases for the C++ exception mechanism is to throw more expressive locally defined types refined from public types, and for catch clauses to catch the most expressive type it knows about, this made dynamic exception specifiers useless in real world code. They also added extra runtime overhead, as they were checked at runtime. For all these reasons, they were removed from the C++ 17 standard, and rightly so.

The difference with `throws(T)` is that `T`'s compatibility is checked at *compile time*, not at run time. It would be required that if `foo() throws(X)` calls `boo() throws(Y)`, that `Y` be convertible to `X`, otherwise it will not compile. This, in combination with the value-based rather than type-based polymorphism explained below, makes these 'static exception specifications' a very different thing to dynamic exception specifications[3].

SG14's proposed design for `std::error` in [P1028] *SG14 `status_code` and standard error object for P0709 Zero-overhead deterministic exceptions* leverages these 'static exception specifications'. Under this, `std::error` is merely one of a family of possible `status_code` types, all of whom can speak to one another, and which can implicitly decay themselves into a `std::error` on demand. Thus, if

[3]Note that `throws(E1, E2, ...)` is explicitly NOT supported for proposed static exception specifications. Any failure polymorphism must be encoded via the *value*, not via the *type*, same as with the proposed C `fails(E)` which exactly mirrors proposed C++ `throws(E)`. This may seem harsh, but note that P1028 provides a bundled value-based polymorphism infrastructure in the standard library.

this true value-based direction were chosen by WG21 instead of some token-based mechanism, all types `T1` in a custom `throws(T1)` would be required to be convertible into the caller's `throws(T2)` (which would usually be `std::error`), and be trivially copyable or move relocating.

One might wonder what the gain would be to permit custom `throws(T)` when it is required that `T` will decay to (usually) `std::error` on demand? The main use case is for very high performance code where constructing a `std::error`, with its probable unavoidable indeterminacy[4], is too unpredictable or expensive. A piece of code may therefore use a custom `throws(T)` perhaps with added (low latency) payload locally, and if the failure is not handled locally, then and only then is a `std::error` lazily constructed, perhaps requiring dynamic memory allocation to store the added payload. This keeps non-deterministic failure handling away from high performance code, and you can find an example of this design pattern in action in [P1031] *Low level file i/o*. It is analogous to how `fails_errno` described earlier works by hoisting the expensive operation up to the caller or the caller's caller.

Another key part of P1028 is *semantic comparison* where it is possible to compare any arbitrary `std::error` value to any other, and equivalence is true when they semantically, rather than literally, match. This allows an implementation to fail with some platform-specific error code, perhaps even with a non-public internal domain, retaining all the original information unmodified and untranslated, yet code further up the call stack can compare the `std::error` value to say `std::errc::no_such_file_or_directory`, and if the platform-specific failure is one of not finding a file or directory, the comparison will return true. Usefully, the programmer may decide to dynamically allocate memory for the payload and store the pointer to that allocation in the code field, and use a custom explanatory domain which 'unpacks' that payload as needed in the semantic comparison. Thanks to semantic comparison, one doesn't need to care about implementation details – one just compares `std::error` values, and it all 'just works'.

As one can infer, this is value-based polymorphism rather than type-based polymorphism, and P1028's semantic comparison mechanism always executes in constant time rather than the unknowable time of a RTTI search. Instead of placing indeterminism into opaque data structures, it exposes the full implementation of deterministic exceptions to the developer, permitting them to choose where and when to introduce dynamic memory allocation or other sources of non-determinism. This is why this alternative mechanism is SG14's preferred fixed latency choice of implementation for P0709.

3.0.6 Background: `std::error_code`

If the P1028 alternative mechanism sounds a bit familiar to some people, it is because it is a superset of the very well understood `std::error_code` which has shipped in the C++ standard library since C++ 11, and well before that in the form of Boost.System. P1028 `status_code` has the following improvements over `std::error_code`:

1. Can represent warnings and informational codes, as well as failure codes.

[4]In order to implement `std::uncaught_exceptions()`, one would have to increment a cold cache thread locally stored reference count. Debuggers probably would also need to be triggered, as all mainstream debuggers offer the ability to break on exception throw. Some runtimes may also wish to capture a stack backtrace.

2. Ambiguities and some legacy cruft in some parts of `std::error_code` have been eliminated e.g. `std::error_code` provides both semantic and literal comparisons via `operator==`, and it is not obvious from inspection of code which is in play in a given piece of code. P1028 always uses semantic comparisons between dissimilar code domains, thus eliminating the ambiguity.
3. Safe to use custom domains in header only libraries which are included into shared libraries, unlike custom categories for `std::error_code` where semantic comparison randomly stops working silently, thus making custom categories unsafe in header only libraries.
4. Domain defined code and payload type, rather than the sole `int` which `std::error_code` provides. Indeed, P1028 wraps all possible `std::error_code`'s into itself by a domain which defines a code type of `std::error_code`, see below.
5. Reduced runtime overhead through maximum use of constexpr, including the new constexpr virtual functions added to C++ 20.
6. Backwards compatible with `std::error_code`, in that any `std::error_code` can be wrapped into a `status_code` with complete preservation of the original code, and with its semantic comparison mapped into `status_code`'s semantic comparison.
7. Including its implementation does not drag in most of the standard library as a dependency, as is currently the case for `std::error_code`.
8. Prewritten domains are supplied for generic, POSIX, Win32, NT kernel, Microsoft COM and `std::error_code` codes, with pregenerated semantic comparison mappings between each.

3.0.7 Background: `std::exception_ptr`

C++ 11 introduced a smart pointer `std::exception_ptr` which refers to a previously thrown C++ exception object. It can be rethrown, examined later, etc. Under deterministic exceptions, if implemented via the SG14 proposed `std::error`, `std::exception_ptr` would become a pointer to a type-based exception throw only[5], and could be used to convert a non-deterministic type-based exception throw into a deterministic value-based exception throw via wrapping the legacy throw into a `std::exception_ptr`, and transporting that via a throw of `std::error`.

P1028, when combined with P1029 *SG14 [[move_relocates]]* or equivalent, permits `std::exception_ptr` instances to be thrown directly, wrapped into a custom domain which provides the semantic mapping of whatever is stored by the exception pointer against other `std::error` instances. Thus, a legacy exception throw of a complex type could be erased into a `std::exception_ptr`, thrown upwards unwinding the stack using the value-based lightweight system, and caught in a catch handler exactly as at present. The semantic comparison feature works exactly as expected with `std::exception_ptr` instances, if they are an instance of a standard library exception type. This can provide perfect source compatibility with existing code, but eliminating EH table bloat entirely in executables for those C++ users happy to forgo binary compatibility with binaries built by earlier C++ compilers.

[5]There is no point erasing types compatible with SG14 proposed `std::error` into an `std::exception_ptr`, as they are not an exception type. Throwing a type convertible into SG14 proposed `std::error` is actually *control flow*, and only the throw of `std::error` itself is an exception throw. I will try to write a paper specifically on this topic soon, hopefully before the Kona meeting.

It is expected that most compiler vendors would provide a compile time switch which lets end users choose between EH table bloat with backwards binary compatibility, or no EH table bloat and loss of backwards binary compatibility.

3.0.8 How C++ function linkage would work using proposed `fails(E)`

We have established that whether token or value based, the proposed `std::error` object would be trivially copyable or move relocating, and fit into no more than two CPU registers. As explained above, this means that a `std::error` instance can safely pass through C code, and may often be entirely legal for C to construct and manipulate, depending on its domain.

I therefore propose that for all C++ functions with a `throws(X)` modifier indicating that they throw *values* of type `X` using the lightweight mechanism, these would map onto a function with a C representation and calling convention of `fails(X)`.

If the modifier is without type i.e. `throws`, that would map onto a function with a C representation of `fails(struct cxx_std_error)`, which would be defined as:

```
struct cxx_std_error
{
  void *domain;
  intptr_t code;
};
```

The above structure is exactly the layout and composition of SG14's proposed `std::error` (which has standard layout, and so can be cast to the above C structure legally in C++).

C++ functions marked `noexcept` would always map onto non-`fails` C representations.

And finally, C++ functions with legacy C++ throw potential would be down to compiler switches to choose form of calling convention, as per the discussion of potential implementation options in P0709.

As one can see, this makes the only difference between C `fails(X)` and C++ `throws(X)` one of *auto-propagation*. Not explicitly handling calling a C `fails(X)` function by using either `try(function(...))` or `catch(function(...))` is a compile time error, whereas not explicitly handling calling a C++ `throws(X)` function means that the compiler silently inserts an as-if `try(expr)`, thus causing unhandled failure to auto-propagate to the caller.

It is proposed that in C++, you can call a C++ `throws(X)` function with C `try()` or C `catch()`, if you would like (note that in C++, `try()` and C `catch()` can accept complex expressions, whereas in C they only accept just the function call).

It is proposed that you can also call a C `fails(X)` function from C++ *without* a C `try(expr)` or C `catch(expr)`. This would cause failure by a C function to manifest as if a deterministic C++ exception of type `X` had been thrown, where the type `X` must be convertible into the calling function's `throws(Y)` (see below for how to handle calls of `fails_errno` C functions from C++).

Given this equivalence mapping of `fails(X)` and `throws(X)` onto one another, we can demonstrate some examples of C++ to C linkage equivalences:

C++	C
`extern int func(double) throws;`	`extern int _Z4funcd(double) fails(struct cxx_std_error);`
`extern int func(double) throws(char *);`	`extern int _Z4funcd(double) fails(char *);`
`extern double func(int) noexcept;`	`extern double _Z4funci(int);`

3.0.9 C++ calling `fails_errno` C functions

P1028 provides a specific constructor of `std::error` for POSIX error codes, `posix_code(int)`. So I would propose that for C++ functions marked `throws` or `throws(E)` which call a `fails_errno` function which fails, the returned code is automatically propagated as a throw of `std::error` constructed by `posix_code(errno code)`.

I would love if type-based exception throwing C++ functions and `noexcept` C++ functions could do the same. However, for existing source compatibility, they cannot. I therefore propose that for those kinds of C++ function, the real `errno` setting avoidance technique described in section 2.1.2 is used.

A number of people have asked how a C++ `throws` function would go about calling a C `fails_errno` *without* it throwing a deterministic exception. The answer is 'exactly as how you would with any other function' e.g.:

```
extern "C" fails_errno int myabs(int x);
...
std::string func(int x) throws
{
  std::expected<int, std::error> r = catch(myabs(x));
  return r ?
    std::to_string(r.value()) :
    std::string("failed to myabs due to ") + r.error().message();
}
```

4 Frequently Asked Questions

4.1 How interchangeable for one another are the proposed C and C++ facilities?

C `fails(E)` can be exchanged for C++ `throws(E)`, with the only difference being that calling a `fails(E)` function does not auto-propagate failure to its caller, and thus requires wrapping its invoking expression with either `catch(...)` or `try(...)`.

C `return failure(expr);` can be swapped for C++ `throw expr;`, but only in a `throws(E)` function.

C++ `std::expected<A, B>` would gain the ability to implicitly construct from the aggregate initialiser which `catch()` expands into.

C `catch(function(...))` would when used in C++ permit the more generalised `catch(expr)`.

C `try(function(...))` would when used in C++ also permit the more generalised `try(expr)`.

5 Acknowledgements

On SG14 Low Latency study group, thanks to JF Bastien, Paul Bendixen, Ben Craig, Odim Holmes, Nevin Liber, Vicente J. Botet Escriba, Arthur O'Dwyer, Herb Sutter and Tony van Eerd for feedback and suggestions.

On WG14, thanks to David Keaton, Jacob Navia, Derek M Jones, Jens Gustedt, Florian Weimer, Martin Uecker, Joseph Myers and Doug Gwyn for suggesting a complete rearchitecture of this proposal into its current form.

On std-proposals, thanks to Nicol Bolas, Thiago Macieira, Andrew Giese, Marcin Jaczewski, Russell Johnston, Bengt Gustafsson and David Brown for feedback and suggestions.

I apologise if I missed your name in the above. There has been an awful lot of feedback from multiple forums. Omissions gladly rectified upon notification.

6 References

[N3051] Doug Gregor,
Deprecating Exception Specifications
https://wg21.link/N3051

[P0323] Vicente J. Botet Escribá, JF Bastien,
std::expected
https://wg21.link/P0323

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[P0786] Vicente J. Botet Escribá,
ValuedOrError and ValueOrNone types
https://wg21.link/P0786

[P0829] Ben Craig,
Freestanding proposal
https://wg21.link/P0829

[P0939] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
http://wg21.link/P0939

[P1028] Niall Douglas,
SG14 `status_code` and standard error object for P0709 Zero-overhead deterministic exceptions
http://wg21.link/P1028

[P1029] Niall Douglas,
SG14 [[move_relocates]]
http://wg21.link/P1029

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031