Introduction to proposed std::expected<T, E>

Niall Douglas

Contents:

1. Relevant WG21 papers this talk is based on
2. Why might we need Expected?
3. What is the current proposed Expected?
4. Use in functional programming
5. Proposed expected operators
6. Implementing object construction with Expected
7. Problems with Expected (in my opinion)

Relevant WG21 papers

Relevant WG21 papers

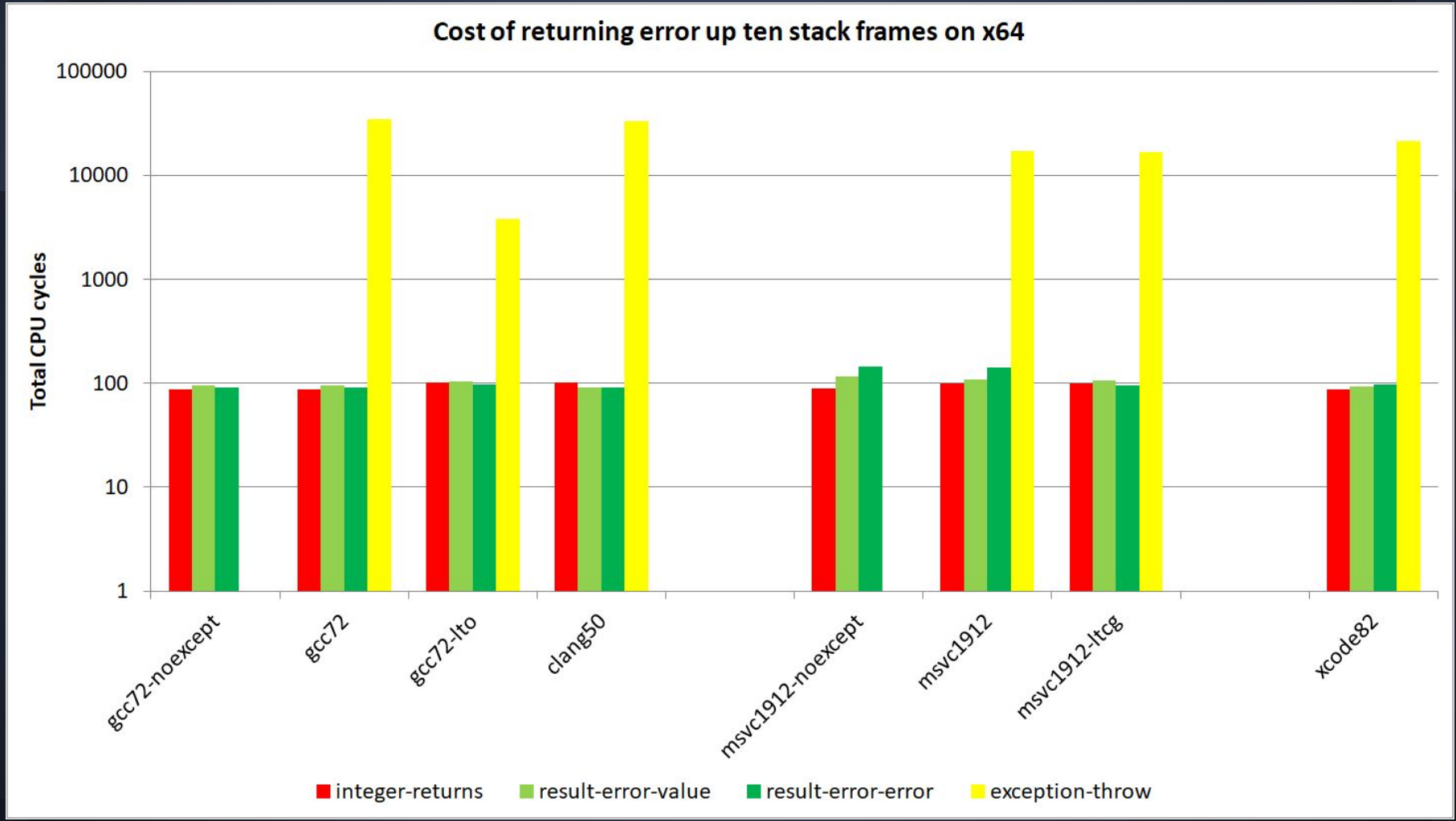
This talk is based on these specific papers:

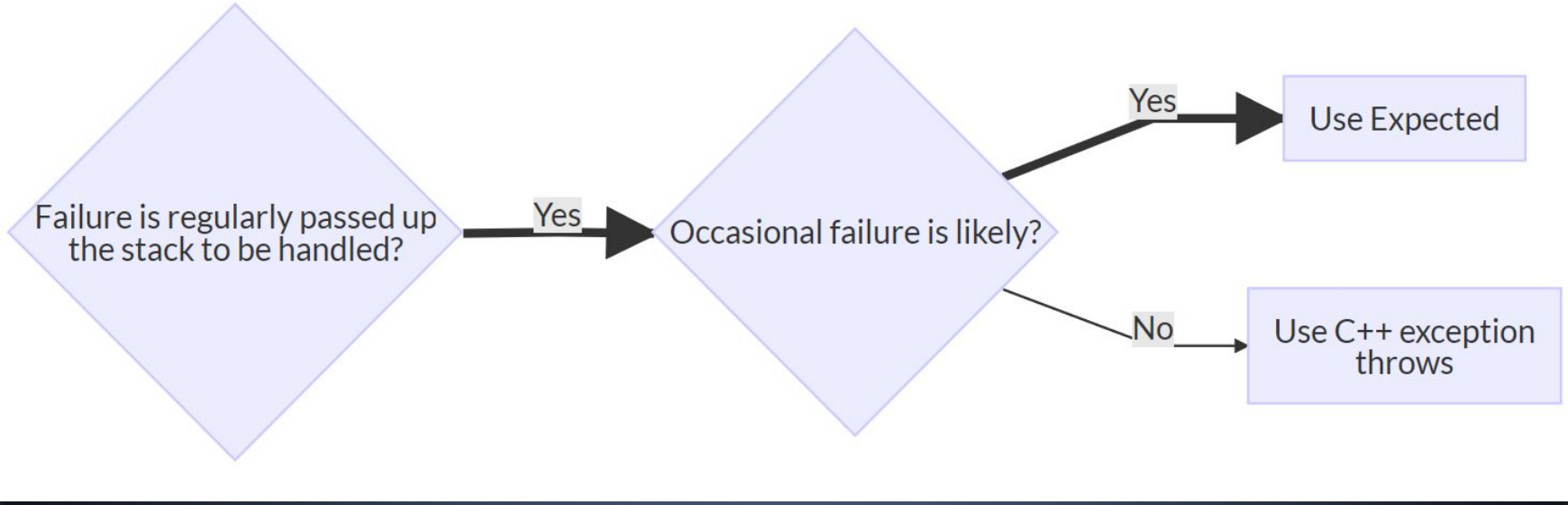
1. P0157R0 *Handling Disappointment in C++* (Lawrence Crowl, 2015)
2. P0323R3 *Utility class to represent expected object* (Vicente J. Botet Escriba, 2014-2017)
3. P0650R1 *C++ Monadic interface* (Vicente J. Botet Escriba, 2016-2017)

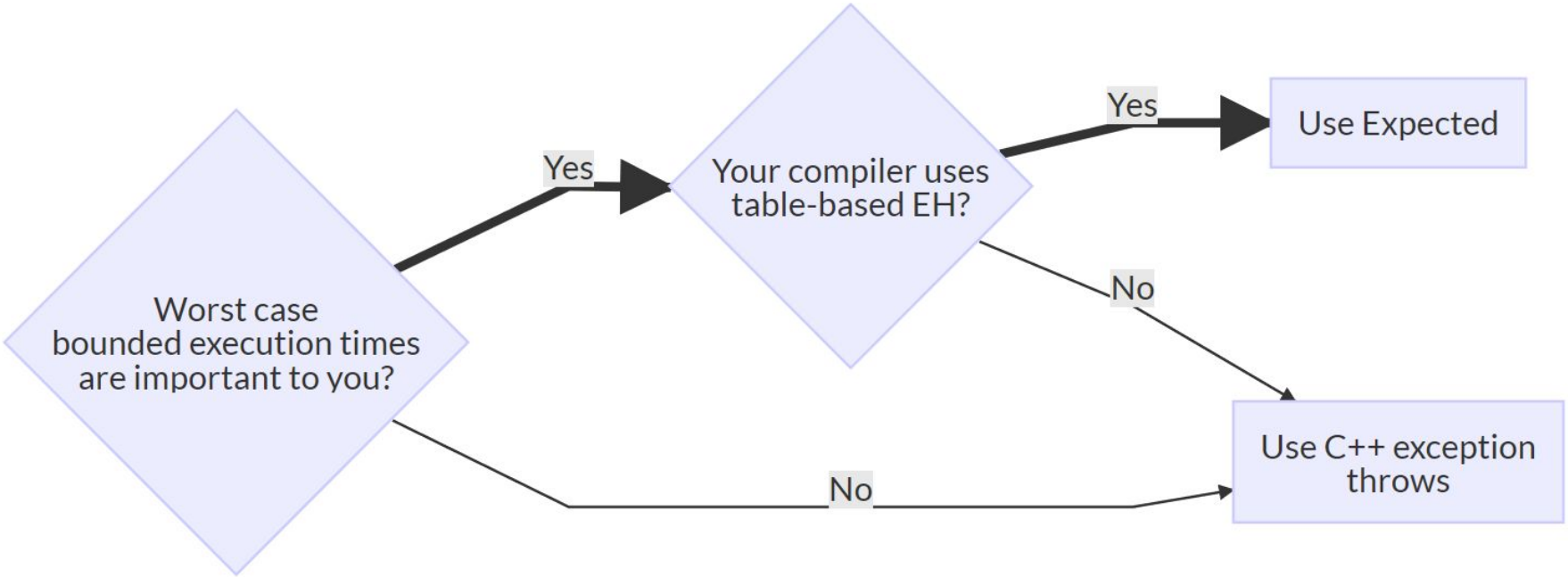
Relevant WG21 papers

4. P0762R0 *Concerns about expected<T, E> from the Boost.Outcome peer review* (Niall Douglas [me!], 2017)
5. P0779R0 *Proposing operator try() (with added native C++ macro functions!)* (Niall Douglas [me!], 2017)
6. P0786R0 *ValuedOrError and ValuedOrNone types* (Vicente J. Botet Escriba, 2017)

Why might you need Expected?


Cost of returning error up ten stack frames on x64
Total CPU cycles
100000
10000
1000
100
10
1
gcc72-noexcept
gcc72
gcc72-lto
clang50
msvc1912-noexcept
msvc1912
msvc1912-ltcg
xcode82
integer-returns
result-error-value
result-error-error
exception-throw


Failure is regularly passed up the stack to be handled?
Yes
Occasional failure is likely?
Yes
Use Expected
No
Use C++ exception throws


Worst case bounded execution times are important to you?
Yes
Your compiler uses table-based EH?
Yes
Use Expected
No
No
Use C++ exception throws

And if you answer yes to any of these questions:

- Failure handling logic is as important or more important than success handling logic?
- The cost of fully testing your code for exception safety isn't worth it to your organisation?
- Code peer review checks correctness of failure handling first?

And if you answer yes to any of these questions:

- You compile with C++ exceptions globally disabled?
- You wish to start adding exception throwing code into an older codebase not written to be exception safe?
  - For example, introducing snazzy new classes from the C++ 17 STL into a mature Qt codebase

Questions?

What is the current proposed Expected?

P0323R3 *Utility class to represent expected object*
(Vicente J. Botet Escriba, 2014-2017)

`expected<T, E>`

Design-wise the proposed `expected<T, E>` sits in between C++ 17's `std::optional<T>` and `std::variant<...>`

- Like a variant, stores either a `T` or an `E`, but with strong "never empty" guarantees
- But has the API of an optional with a `T` state being an "expected" thing and an `E` state being an "unexpected" thing

```
template<class T, class E>
requires(is_nothrow_move_constructible_v<E> && !is_void_v<E>)
class expected {
public:
  // all the same member functions from optional<T>
  using value_type = T;
  constexpr expected(...);  // implicit usual ways of
constructing a T, usual assignment, swap, etc
  constexpr T* operator ->();
  constexpr T& operator *();
  constexpr explicit operator bool() const noexcept;
  constexpr bool has_value() const noexcept;
```

```
  constexpr T& value();
  template <class U> constexpr T value_or(U&&);

  // with these additions
  using error_type = E;
  constexpr expected(unexpected_type<E>);  // type sugar for
constructing an E
  constexpr E& error();
};

// C++17 template deduced unexpected type sugar
template<class E>
class unexpected { ...
```

Questions?

Example of use

```
enum class arithmetic_errc {
  divide_by_zero,            // 9 / 0 == ?
  not_integer_division,      // 5 / 2 == 2.5 (which is not an integer)
  integer_divide_overflows, // INT_MIN / -1
};

expected<int, arithmetic_errc> safe_divide(int i, int j) {
  if (j == 0)
    return unexpected(arithmetic_errc::divide_by_zero);
  if (i == INT_MIN && j == -1)
    return unexpected(arithmetic_errc::integer_divide_overflows);
  if (i % j != 0)
    return unexpected(arithmetic_errc::not_integer_division);
  return i / j;
}
```

```
expected<double, arithmetic_errc>
  f1(double i, int j, int k)
{
  auto q = safe_divide(j, k);
  // propagate any error
  if (!q)
    return unexpected(q.error());

  // otherwise act on value
  return i + *q;
}
```

Littering your code with these kinda sucks!

Two solutions in the works (both would ship well after Expected):

1. `operator try` (P0779R0)
2. C++ Monadic interface (P0650R1)

Use in functional programming

P0650R1 *C++ Monadic interface*

(Vicente J. Botet Escriba, 2016-2017)

```
functor::transform          : [T] x (T->U) -> [U]
functor::map                : (T->U) x [T] -> [U]

applicative::ap             : [T] x [(T->U)] -> [U]
applicative::pure<A>        : T -> [T]

monad::unit<A>              : T -> [T]
monad::bind                 : [T] x (T->[U]) -> [U]  // mbind
monad::unwrap               : [[T]] -> [T]           // unwrap
monad::compose              : (B->[C]) x (A->[B])-> (A->[C])

monad_error::catch_error    : [T] x (E->[T]) -> [T]
```

```
expected<int, arithmetic_errc> f(int i, int j, int k)
{



  return monad::bind(safe_divide(i, k), [=](int q1) {
    return monad::bind(safe_divide(j,k), [q1](int q2) {
      return q1 + q2;
    });
  });
}
```

Questions?

Proposed Expected operations

P0786R0 *ValuedOrError and ValuedOrNone types*

(Vicente J. Botet Escriba, 2017)

Expected Concepts

P0786R0 proposes two *Concepts*:

1. **`ValueOrError`**
   - Accepts *any* type which provides **`.has_value()`**, **`.value()`** and **`.error()`** e.g. **`expected<T, E>`**

2. **`ValueOrNone`**
   - Accepts *any* type which provides **`.has_value()`** and **`.value()`** e.g. **`optional<T>`**

Expected operators

You then get these standard operations:

- `operator try` --OR-- `operator ?`
- `value_or`
- `value_or_throw`
- `error_or`
- `check_error`

These let you save typing boilerplate ...

try/?

```
expected<T, E> e1 = expr1(...);
if(!e1.has_value())
  return e1.error();
T v1 = std::move(e1.value());
```

```
// If operator try proposal:
T v1 = try expr1(...);
// If operator ? proposal then like Swift:
T v1 = expr1?(...);
```

value_or

```
template <ValueOrNone X, class T>
auto value_or(X&& x, T&& v) {
  return x.has_value() ? x.value() : v;
}
```

```
expected<T, E> e1 = expr1(...);
T v0 = ...;
T v1 = value_or(e1, v0);
```

value_or_throw

```
template <class E, ValueOrNone X>
auto value_or_throw(X&& x) {
  return x.has_value() ? x.value() : throw E{};
}
template <class E, ValueOrError X>
auto value_or_throw(X&& x) {
  return x.has_value() ? x.value() : throw E{x.error()};
}
```

```
expected<T, std::error_code> e1 = expr1(...);
T v1 = value_or_throw<system_error>(e1);
```

value_or

```
template <ValueOrError X, class T>
auto error_or(X&& x, T&& v) {
  return !x.has_value() ? x.error() : v;
}
```

```
expected<T, E> e1 = expr1(...);
E v0 = ...;
E v1 = error_or(e1, v0);
```

check_error

```
template <ValueOrError X, class E>
bool check_error(X&& x, E&& err) {
  if(x.has_value())
    return false;
  return x.error() == err;
}
```

```
expected<T, E> e1 = expr1(...);
E v0 = ...;
if(check_error(e1, v0))
  ...
```

Questions?

Implementing object construction with Expected

100% Expected based constructors

- I get asked about this *a lot* … so here is one of many possible solutions
  - Developed during implementation of my proposed File I/O TS hopefully going before WG21 in 2018
- Prerequisites:
  - Your type must be movable
  - Moves must be cheap
  - You don’t mind a little bit of metaprogramming
  - You don’t mind typing more characters

100% Expected based constructors

Steps:

1. Break construction into two phases:
   a. An all-constexpr phase which places the object into a valid, legally destructible state
   b. All operations which aren't constexpr and/or could fail go into the second, Expected-returning, phase
2. Tell the metaprogramming how to construct your object
3. End users now do `construct<Foo>{Args...}()`

Phase 1: All-constexpr constructor

```
class file_handle {
  int _fd{-1};  // file descriptor
  struct stat _stat {
    0
  };  // stat of the fd at open

  // Phase 1 private constexpr constructor
  constexpr file_handle() {}

  // Phase 2 static member constructor function, which cannot throw
  static inline expected<file_handle, std::error_code> file(path_type path, mode
mode = mode::read) noexcept;
};
```

Phase 2: Expected returning stage

```
// Phase 2 static member constructor function, which cannot throw
inline expected<file_handle, std::error_code>
file_handle::file(file_handle::path_type path, file_handle::mode mode) noexcept
{
  // Perform phase 1 of object construction
  file_handle ret;
  ...
  // Perform phase 2 of object construction
  ret._fd = ::open(path.u8string().c_str(), flags);
  if(-1 == ret._fd)
    return unexpected(error_code{errno, std::system_category()});
  ...
  return {std::move(ret)};
}
```

Defining construct<T>

```
template <class T> struct construct {
  void operator()() const noexcept {
    static_assert(!std::is_same<T, T>::value, "construct<T>() was not
specialised for the type T supplied");
  }
};
template <> struct construct<file_handle> {
  file_handle::path_type _path;
  file_handle::mode _mode{file_handle::mode::read};
  // Any other args, “default initialised” if necessary, follow here ...

  expected<file_handle, std::error_code> operator()() const noexcept { return
file_handle::file(std::move(_path), _mode); }
};
```

Why this design?

Usage is thus: `construct<file_handle>{"hello"}();`

Why this choice of design?

- Passing around empty-callable objects is very useful for metaprogramming
- Nothing wrong with `{}(Args&&...)` of course ...
- But free functions are inferior to type specialisation
  - Free functions cannot be partially specialised, so tag dispatch is needed for those
  - Which is more typing, and much slower to compile

Questions?

Problems with Expected

P0762R0 *Concerns about expected<T, E> from the Boost.Outcome peer review*

(Niall Douglas [me!], 2017)

Expected problems (in my opinion):

1. It takes too much typing to use! Too verbose!
2. It alone doesn't solve alone the dual-`error_code`-API problem
3. The fact it can throw exceptions adds no value and adds significant costs to developers
    - Forces exception awareness onto devs

Expected problems (in my opinion):

4. Lack of stable ABI guarantees precludes usage in public interfaces for big iron C++ users
   - Specifically, no C compatibility
5. Underspecified on what happens at ABI boundaries where library A's custom Expecteds meet library B's?

1. The verbosity

```
struct handle;      // Abstract base class for some handle implementation
class handle_ref;   // Some sort of smart pointer managing a handle *
// Returns the expected opened handle on success, or an
// unexpected cause of failure
extern std::expected<handle_ref, std::error_code> openfile(
        const char *path) noexcept {
  int fd = open(path, O_RDONLY);
  if(fd == -1) {
    return std::unexpected(std::error_code(errno, std::system_category()));
  }
  std::error_code ec;
  auto *p = new(std::nothrow)
    some_derived_handle_implementation(fd, ec);
```

```
  if(p == nullptr) {
    close(fd);
    // C++ 11 lets you convert generic portable error_condition's
    // into a platform specific error_code like this
    return std::unexpected(std::make_error_code(std::errc::not_enough_memory));
  }
  // The some_derived_handle_implementation constructor failed
  if(ec) {
    delete p;
    return std::unexpected(std::move(ec));
  }
  return handle_ref(p);  // expected<> takes implicit conversion
                         // to type T
}
```

```
std::expected<handle_ref, std::error_code> fh_ = openfile("foo");
// C++ 11 lets you compare some platform specific error code to a
// generic portable error_condition
if(!fh_ && fh_.error() != std::errc::no_such_file_or_directory) {
  if(fh_.error() == std::errc::not_enough_memory) {
    throw std::bad_alloc();
  }
  ... more ...
  // If unhandled, abort by throwing a system_error wrapping the code
  throw std::system_error(std::move(fh_.error()));
}
if(fh_) {
  handle_ref fh = std::move(fh_.value());
  fh->read(... etc
}
```

2. The dual error code API problem

```
namespace filesystem {
  /*! Copies the file at path `from` to path `to`.
  \returns True if file was successfully copied.
  \throws On failure throws `filesystem_error(ec.message(), from, to, ec)`
  with `ec` being the error code reported by the operating system.
  */
  bool copy_file(const path &from, const path &to);

  /*! Copies the file at path `from` to path `to`.
  \returns True if file was successfully copied. If false, `ec` is written
  with the error code reported by the operating system.
  \throws Never throws.
  */
  bool copy_file(const path &from, const path &to, std::error_code &ec)
noexcept;
}
```

3. The forced exception awareness problem

Expected's throwableness

1. **`.error()`**, **`operator->`** and **`operator*`** are *narrow* observers
   - Loss of program correctness (i.e. hard UB)
2. **`.value()`** alone is a *wide* observer
   - Can throw **`bad_expected_access<E>`**

What value does a throwing .value() add?

If the user really wants that, it is ***trivial*** to add on top

4. Lack of stable ABI guarantees

Big iron C++ interface requirements

1. Components are ABI stable over many years
   - Avoids needing to recompile components
   - Often enforced by automated tooling
   - Prevents ripples of change affecting other teams
2. Interface files have near-zero compile time impact and don't **`#include`** much else
   - Really matters with millions of compilands
   - Reduces header dependency management

Consequences

- Use of the STL is usually banned in public APIs
- Public APIs usually cannot throw exceptions
  - Indeed, are often C-compatible
- Examples:
  - Microsoft COM (no STL, C compatible)
  - Qt (no STL, no non-fatal C++ exception throws)
  - Google C++ style guide (bans much of the STL, most of Boost, bans C++ exception throws)

Will Expected be allowed into public APIs? **No!**

5. ABI interop underspecified

```
namespace lib_A {
  // Error code + paths related to a failure.
  struct failure_info {
    std::error_code ec;
    path path1, path2;
  };
  expected<void, failure_info>
write_file(std::string_view data) noexcept;
}
```

```
namespace lib_B {
  enum class status_code {  // HTTP status codes
    bad_request = 400,
    ...
  };
  struct failure {
    status_code status{status_code::success};
    std::string url{};  // The failing URL
  };
  // Performs a HTTP GET on the url, calling
  // completion(std::expected<std::string,failure>)
  // with results.
  template<class F>
  expected<void, failure> get(std::string url, F&& completion);
}
```

```
namespace some_app {
  lib_B::get("http://meetingcpp.com/",
    [](expected<std::string, lib_B::failure> contents)
      -> expected<void, lib_B::failure> {
        if(!contents)
          return unexpected(contents.error());
        expected<void, lib_A::failure_info> r =
          lib_A::write_file(*contents);
        if(r)
          return {};  // success
        ??? // How do I return failure to write the file?
    });
}
```

Thank you

And let the questions begin!