Mongrel Monads

Dirty, dirty, dirty

Niall Douglas

Contents:

1. Four techniques for handling errors in modern C++
   - Integer return codes [C]
   - Exception throws [C++ 98]
   - Error codes (`std::error_code` et al) [C++ 11]
   - Mongrel monads!
     - `std::optional<T>` [C++ 17]
     - `std::expected<T, E>` [C++ 20 ?]

(A proposed Boost.Outcome library up for peer review end of May brings you the proposed `expected<T, E>` for standardisation today for any C++ 14 compiler)

Contents:

2. Benchmarking the error handling techniques on GCC 6.2, clang 4.0 and VS2017
3. Proposed Boost.Outcome's convenience extensions to the WG21 proposal
    - `option<T>`, `result<T>` and `outcome<T>`
4. Mongrel monads by code example

C error handling

Integer return codes

C Integer return codes

- Define some <u>*domain*</u> of integer error codes meaning different types of error
- Variants:
  - Use an `enum` rather than macros to represent the domain (slightly more type safety)
  - Return the integer from a function directly
  - Return it via a thread local facility such as `errno` or `GetLastError()`
  - Return it via an `int*` in the parameter list

```c
struct handle {
  int fd;
  ...
};
enum errors {
  SUCCESS=0,
  NOMEM,
  NOTFOUND,
  ...
};
```

```c
extern int openfile(struct handle **outh, const char *path) {
  *outh = malloc(sizeof(struct handle));
  if(!*outh)
    return NOMEM;
  (*outh)->fd = open(path, O_RDONLY);
  if((*outh)->fd == -1) {
    free(*outh);
    *outh = NULL;
    return NOTFOUND;
  }
  return SUCCESS;
}
```

C++ 98 error handling

Throwing exceptions like it’s 1998

C++ 98 Exception throws

- If an error occurs, throw an exception to indicate the problem
- Often misused to indicate input parameters have bad values etc
- More subtle misuse is as *control flow* where better alternatives exist
- Can be really expensive, as we will see later

```
// Abstract base class for some handle implementation
struct handle {
  int fd;
  ...
  virtual ~handle() {
    if(fd != -1) {
      close(fd);
      fd = -1;
    }
  }
};
class handle_ref;  // Some sort of smart pointer managing a
handle *
```

```
extern handle_ref openfile(const char *path) {
  int fd = open(path, O_RDONLY);
  if(fd == -1) {
    throw std::runtime_error("File not found");
  }
  // RAII close the file if exception throw
  handle temp(fd);
  // Could throw std::bad_alloc
exception during construction
  return handle_ref(new some_de
implementation(temp));
}
```

This code is C++ 98

Can anyone say what exception type should be thrown here instead in C++ 11?

C++ 11 error handling

The underutilised C++ 11 **<system_error>**, and what does **noexcept** actually mean?

C++ 11 error codes

- C++ 11 brought in Boost.System as the <system_error> header
  - Provides a C++ equivalent to C integer error codes
    - A singleton subclass of `std::error_category` provides the domain (i.e. what the codes mean)
    - `std::error_code` is an integer and a reference to some `error_category` instance
    - `std::system_error` subclasses `std::exception` to transport a `std::error_code` as *payload*

C++ 11 error codes

- Not widely used in C++ 11 nor C++ 14 standard libraries which currently remain exception throw heavy
  - BUT C++ 17's `<filesystem>` uses `error_code` throughout
  - As does the Networking TS (ASIO)
  - Expect to see new overloads using `error_code` cropping up in future C++ standard libraries

C++ 11 error codes

- The cleverness of **`system_error`** is not widely appreciated! (1)
  - Implements framework for testing *semantic equivalence* between error codes
    - This means you do not need to “translate” one error code domain into another with **`switch()`**

```
// Make a system-specific error code matching this error condition
std::error_code ec(std::make_error_code(std::errc::timed_out));

// Compare some system-specific error code to this error condition
if(ec == std::errc::not_enough_memory) ...
```

C++ 11 error codes

- The cleverness of **`system_error`** is not widely appreciated! (2)
  - Lets you “wrap” any existing C integer error code system without having to recompile that C library!
  - Out of the box **`system_error`** provides default domains for POSIX **`errno`** and Win32 **`GetLastError`**()

C++ 11 noexcept

- C++ 11 also brought us the **noexcept** modifier to indicate that calling a function will <u>*never*</u> throw an exception
- But what does **noexcept** mean?
  a. That this function cannot return an error? Maybe
  b. That the optimiser can assume that calling this function can only exit through normal return? Yes
  c. That this function calls `std::unexpected()`? No
  d. That this function calls `std::terminate()`? Maybe

```
struct handle;      // Abstract base class for some handle
implementation
class handle_ref;  // Some sort of smart pointer managing a handle
*

// Non-throwing overload
extern handle_ref openfile(const char *path, std::error_code &ec)
noexcept {
  int fd = open(path, O_RDONLY);
  if(fd == -1) {
    // Construct an error code in the OS errors domain
    ec = std::error_code(errno, std::system_category());
    return {};
  }
  auto *p = new(std::nothrow)
some_derived_handle_implementation(fd, ec);
```

```cpp
  if(p == nullptr) {
    close(fd);
    // Construct an error code matching the generic OS error
    // equivalent to the ENOMEM error condition
    ec = std::make_error_code(std::errc::not_enough_memory);
    return {};
  }
  if(ec) {
    delete p;
    return {};
  }
  return handle_ref(p);
}
```

```cpp
struct handle;      // Abstract base class for some handle implementation
class handle_ref;   // Some sort of smart pointer managing a handle *
// Non-throwing overload
extern handle_ref openfile(const char *path, std::error_code &ec) noexcept {
  int fd = open(path, O_RDONLY);
  if(fd == -1) {
     // Construct an error code in the OS errors domain
     ec = std::error_code(errno, std::system_category());
     return {};
  }
  auto *p = new(std::nothrow) some_derived_handle_implementation(fd, ec);
  if(p == nullptr) {
     close(fd);
     // Construct an error code matching the generic OS error equivalent
     // to the ENOMEM error condition
     ec = std::make_error_code(std::errc::not_enough_memory);
     return {};
  }
  if(ec) {
     delete p;
     return {};
  }
  return handle_ref(p);
}
```

You need an ec.clear(); here

Did anyone see a bug in this code?

```
// Non-throwing overload defined on previous page
extern handle_ref openfile(const char *path, std::error_code
&ec) noexcept;

// Throwing overload
extern handle_ref openfile(const char *path)
{
  std::error_code ec;
  handle_ref ret(openfile(path, ec));
  if(ec)
  {
    throw std::system_error(ec);
  }
```

```cpp
  return ret;
}

// If I want an exception throw due to failure to open the file:
auto handle = openfile("somepath.txt");
// If I want to handle failure to open the file in normal control flow:
std::error_code ec;
auto handle = openfile("somepath.txt", ec);
if(ec)
  handle_error(ec);
```

Questions?

C++ 20 (?) error handling

Mongrel monads: dirty, dirty, dirty

C++ 20 need for improvement

We have covered three different ways of returning errors in C++, why do we need a fourth way?

1. Forcing every caller to manually declare a `std::error_code` to pass as **&ec** is unnatural, clunky and not very "C++-ish"
2. A throwing and non-throwing overload of every extern function doubles your public API count!

C++ 20 need for improvement

3. It is also error prone
   - Very easy to accidentally forget to check for **ec** after a function returns
   - Very easy to forget to <u>**clear ec**</u> on entry to a function **Remember that bug earlier?**
   - In practice it's even easier to forget than for C integer returns, <u>so errors get lost or misreported frequently</u>

C++ 20 need for improvement

4. The new systems languages Swift and Rust prefer to return errors via a monadic transport of an integer with some error code domain, and C++ needs to compete
5. **std::error_code** cannot be **constexpr** constructed with an error category, and so cannot be used to transport errors in **constexpr** ☹

This has led the WG21 Library Evolution Working Group (LEWG) to propose a C++ equivalent to a Swift/Rust error transporting monad called **expected<T, E>**

LEWG `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 with the same "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

```cpp
template<class T, class E = std::error_condition>
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 oper
  constexpr bool has_value() const noexcept;
```

This is a defect, it should be std::error_code and the Expected in proposed Boost.Outcome deviates from LEWG Expected on this

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

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

// usual make functions
template <class T> constexpr expected<decay_t<T>> make_expected(T&&);
template <class E> constexpr unexpected_type<decay_t<E>>
make_unexpected(E&&);
```

I personally think that the use of decay_t here is a defect (make_expected<const Foo>(Foo) won't work, and that is quite useful sometimes)

Mongrel monads

In terms of monads:

- Maybe monad => `optional<T>`
- Either monad => `expected<T, E>`

(WG21 LEWG has decided to work on the monadic programming API for these in a separate (later) proposal. Outcome provides its own monadic operators API as an extension)

Rust’s Result<T, E> use example, but in C++

`http://rustbyexample.com/std/result.html`

```cpp
// Replicates example usage of Result<T, E> from
// http://rustbyexample.com/std/result.html
namespace checked {
  // Mathematical "errors" we want to catch
  enum class MathError {
    DivisionByZero,
    NegativeLogarithm,
    NegativeSquareRoot
  };
  using MathResult = outcome::expected<double,
MathError>;
```

```cpp
MathResult div(double x, double y) noexcept {
  if(::fabs(y) < FLT_EPSILON) {
    // This operation would fail, instead let's return the
    // reason of the failure wrapped in E
    return outcome::make_unexpected(MathError::DivisionByZero);
  }
  else {
    // This operation is valid, return the result wrapped in T
    return x / y;
  }
}
```

```
  MathResult sqrt(double x) noexcept {
    if(x < 0.0)
      return
outcome::make_unexpected(MathError::NegativeSquareRoot);
    return ::sqrt(x);
  }
  MathResult ln(double x) noexcept {
    if(x < 0.0)
      return
outcome::make_unexpected(MathError::NegativeLogarithm);
    return ::log(x);
  }
}
```

```
double op(double x, double y) noexcept {
  checked::MathResult ratio = checked::div(x, y);
  if(!ratio) {
    std::cerr << "PANIC: MatchResult::DivisionByZero" <<
std::endl;
    std::terminate();
  }
  checked::MathResult ln = checked::ln(*ratio);
  if(!ln) {
    std::cerr << "PANIC: MatchResult::NegativeLogarithm" <<
std::endl;
    std::terminate();
  }
```

```
  checked::MathResult sqrt = checked::sqrt(*ln);
  if(!sqrt) {
    std::cerr << "PANIC: MatchResult::NegativeSquareRoot" <<
std::endl;
    std::terminate();
  }
  return sqrt.value();
}
int main(void) {
  // Will this fail?
  std::cout << op(1.0, 10.0) << std::endl;
  return 0;
}
```

A better use example

Our **`openfile()`** from before,
but using **`expected<T, E>`**

```
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::experimental::expected<handle_ref, std::error_code>
openfile(const char *path) noexcept {
  int fd = open(path, O_RDONLY);
  if(fd == -1) {
    return std::experimental::make_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::experimental::make_unexpected(
std::make_error_code(std::errc::not_enough_memory));
  }
  // The some_derived_handle_implementation constructor failed
  if(ec) {
    delete p;
    return std::experimental::make_unexpected(std::move(ec));
  }
  return handle_ref(p);  // expected<> takes implicit conversion to
type T
}
```

```
auto 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)
{
  // This is serious, abort by throwing a system_error wrapping the
error code
  throw std::system_error(std::move(fh_.error()));
}
if(fh_)
{
  handle_ref fh = std::move(fh_.value());
  fh->read(... etc
}
```

Questions?

Benchmarking performance

Why not just throw exceptions
and save all this hassle?

Caveats benchmarking error handling

- It is surprisingly hard to come up with a representative benchmark for error handling
- On SG14 (the WG21 study group for low latency C++ games dev, financial trading etc) significant efforts to quantify the overhead of C++ exceptions have been made
- The problem is that any synthetic benchmark you choose will be too simple, and any real world code base you modify will have been designed originally around one particular error handling system

Benchmarking error handling

The benchmark presented here is very simple. For each error handling mechanism:

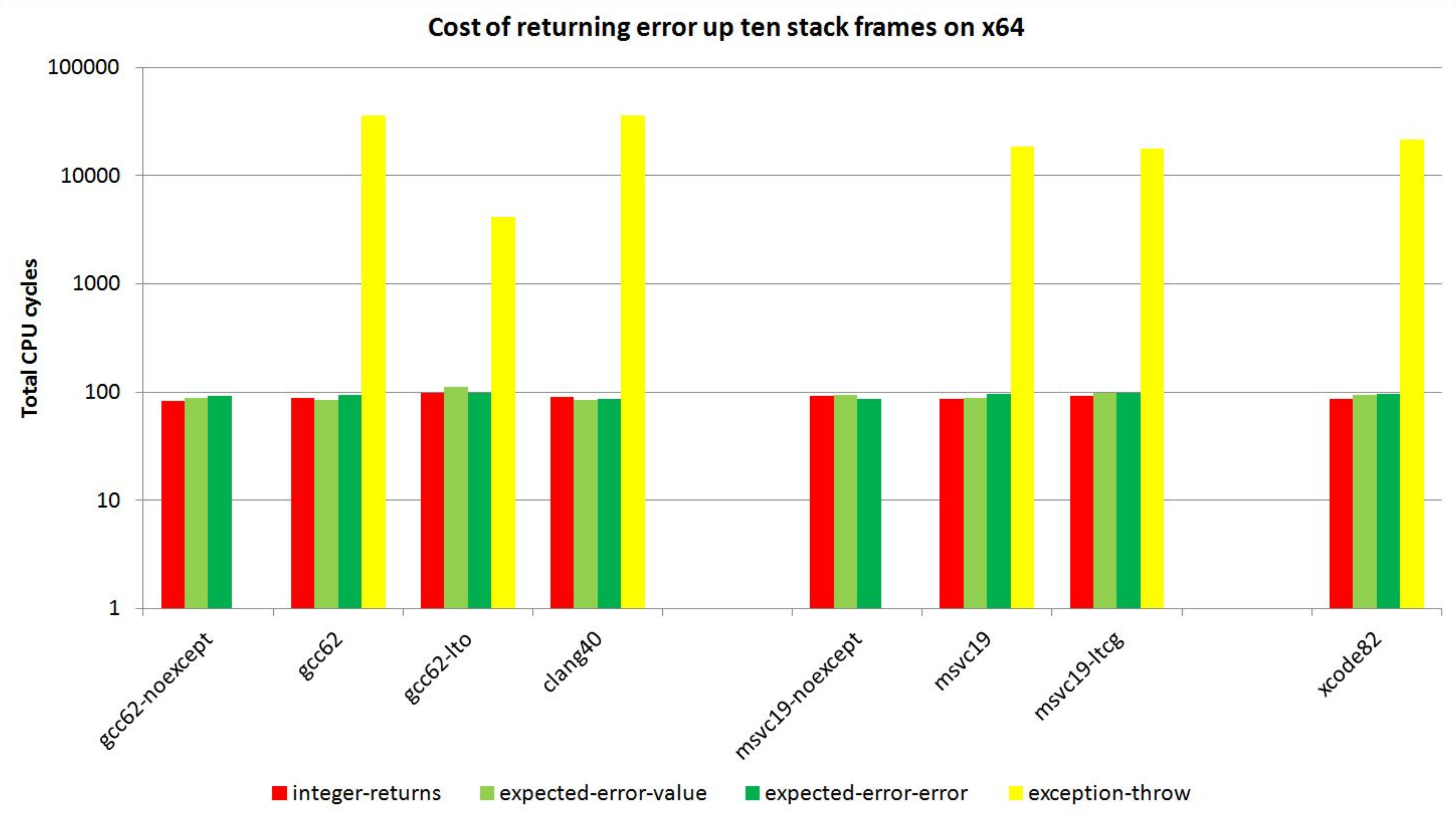
- Generate ten source files each containing a single function which calls the function in the next source file. Access a `volatile int` before and after as work
- Compile each separately and link
- The final function in the call sequence either returns a value or an error
- Iterate 100,000 times for many combinations of compilers and options

Benchmarking error handling

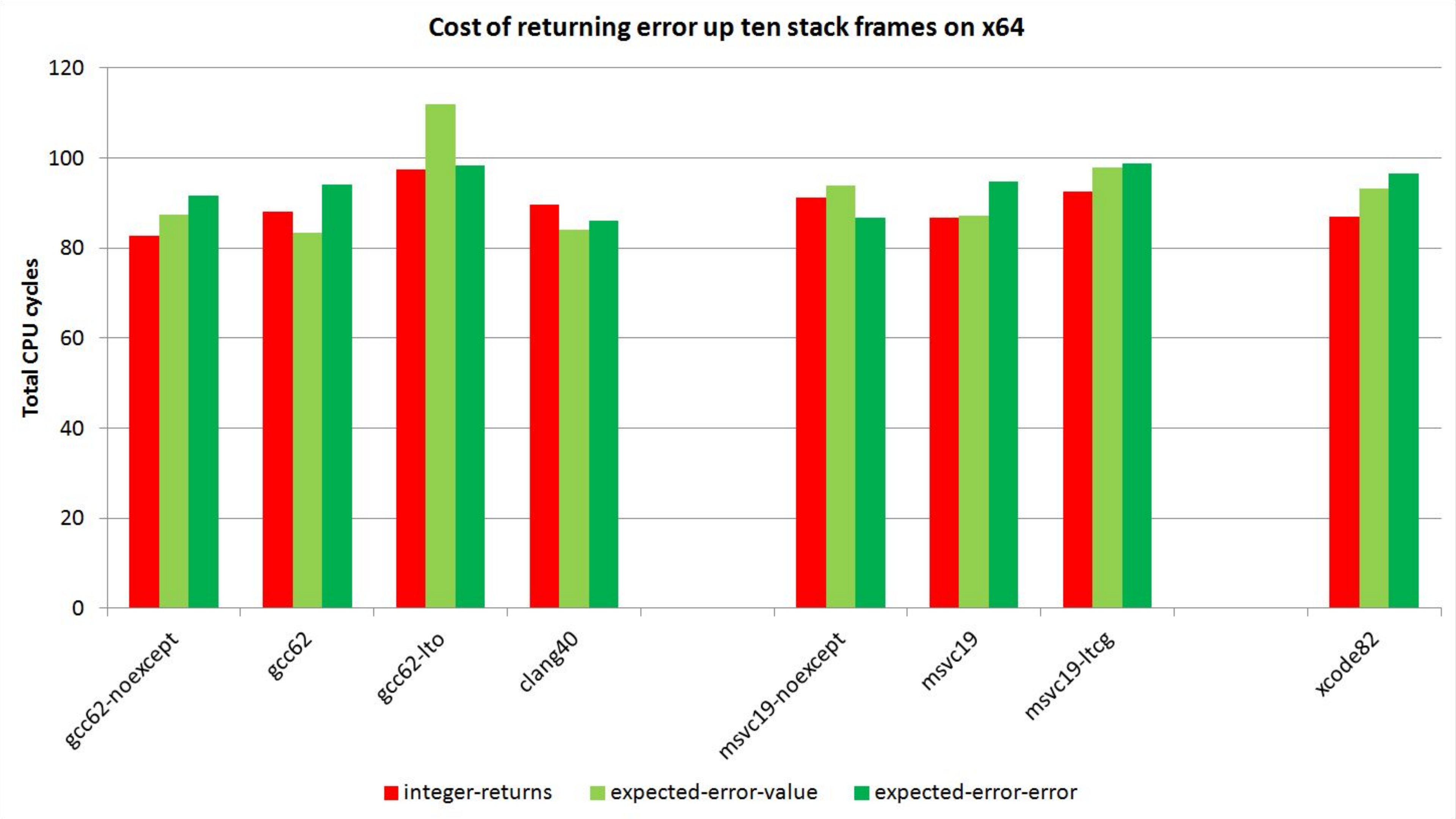
Test hardware (MacBook Pro late 2016):

- 3.1Ghz Skylake CPU
- 25 Gb/sec memory bandwidth with 120 ns main memory latency

- Windows Subsystem for Linux (WSL) running Ubuntu 14.04 LTS
- Microsoft Windows 10 x64 1607
- Apple macOS Sierra 10.12.3



Cost of returning error up ten stack frames on x64
100000
10000
1000
100
10
1
Total CPU cycles
gcc62-noexcept
gcc62
gcc62-lto
clang40
msvc19-noexcept
msvc19
msvc19-ltcg
xcode82
integer-returns
expected-error-value
expected-error-error
exception-throw



Cost of returning error up ten stack frames on x64
Total CPU cycles
0
20
40
60
80
100
120
gcc62-noexcept
gcc62
gcc62-lto
clang40
msvc19-noexcept
msvc19
msvc19-ltcg
xcode82
integer-returns
expected-error-value
expected-error-error

Questions?

Which error handling system should I use?

Which is best?

It really does *depend* on your code …

- Some techniques lend better to some code designs than others
- Throwing exceptions (despite the cost) really can make sense in some designs
  - 30k CPU cycles (~10 µs) is irrelevant compared to operations lasting 10 ms

All that said, `std::error_code` is woefully underused even in brand new C++ code

Proposed Boost.Outcome

Hopefully coming to Boost next month!

Proposed Boost.Outcome

- Comes with a high quality LEWG `expected<T, E>` implementation [*]
    - Compiles into very compact assembler
- Completely standalone:

    <u>* Header only and no Boost needed ***</u>**

- Works well on all major C++ 14 compilers
    - Minimum: clang 3.5, GCC 5, VS2015 Update 2
    - Best: clang 3.6, GCC 7, VS2017

Deviations from LEWG Expected 1

- P0323R1 doesn't yet specify what will be done if you try accessing an expected which is valueless due to exception. We throw a `bad_expected_access<void>`
- Types `T` and `E` cannot be constructible into one another. This is a fundamental design choice to significantly reduce compile times so it won't be fixed.

Deviations from LEWG Expected 2

- Instead of being its own type, `unexpected_type<E>` is template aliased to an `expected<void, E>` which implicitly converts into any `expected<T, E>`
    - Note our `expected<T, E>` passes the LEWG Expected test suite!
- Our `expected<T, E>` defaults `E` to `std::error_code` rather than to `std::error_condition`
    - The LEWG proposal is almost certainly wrong on this, it should be `std::error_code`

Deviations from LEWG Expected 3

- We don't implement the ordering and hashing operator overloads due to https://akrzemi1.wordpress.com/2014/12/02/a-gotcha-with-optional/. The fact the LEWG proposal does as currently proposed is a defect.
- Our Expected always defines the default, copy and move constructors even if the the type configured is not capable of it. That means `std::is_copy_constructible` etc returns true when they should return false. Reason? It reduces compile times

Other features of Boost.Outcome

- Ridiculously comprehensive “small book” of documentation
- Full validation and conformance test suite
- Adds convenience alternatives to `expected<T, E>` called `outcome<T>`, `result<T>` and `option<T>`
    - These have stable ABI guarantees so are safe for returning from DLL exported functions

Other features of Boost.Outcome

- Adds a full fat monadic programming API plus lots of other useful extensions
- Works great with C++ exceptions disabled
  - SG14 low latency friendly
- Can be used as a "unified error handling system" to deal with multiple error handling systems by third party library dependencies
  - (yes I know only cranks and weirdos propose those … but you don't have to use that part if you don't want to)

outcome<T>, result<T> and option<T>

Outcome’s extensions to expected<T, E>

Outcome extensions

`expected<T, E>` is great, but it’s a general purpose STL *primitive* serving double duty:

1. Where the type of `E` is used to enforce error domain type safety by being a different type per error domain
2. Where the type of `E` is always `std::error_code` because errors arise from many, unknown, sources

For the latter use case you need to type more boilerplate code than is ideal

Outcome extensions

So to save typing boilerplate:

- **option<T>** = empty | T
- **result<T>** = empty | T | error code
- **outcome<T>** = empty | T | error code | exception ptr

Hard coding the possible error types means the API lets you skip typing boilerplate

Outcome extensions

```
expected<int, std::error_code> v =
   make_unexpected<std::error_code>(
    std::errc::timed_out);
try {
  v.value();  // throws
}
catch(const bad_expected_access<
std::error_code> &e) {
  // Extract the error code and rethrow
  // as asystem error which is the most
  // appropriate C++ exception type for
```

What happens if these types get out of sync?

```
outcome<int> v =
  make_errored_outcome(
    std::errc::timed_out);
...
v.value();  // throws a
system_error


outcome<int> v =
  make_exceptional_outcome(
    std::bad_alloc());
...
v.value();  // throws a
bad_alloc
```

Outcome extensions

There is also a simple monadic functional programming DSEL whose design annoys all the purists

- Uses distinct overloads to choose operations rather than distinct operators
- Allows (shock horror!) move semantics which is a big no-no in functional programming (but hey, this is C++!)

Quick demo of mongrel monad logic

With minor puzzles!

```cpp
using namespace BOOST_OUTCOME_V1_NAMESPACE;
error_code_extended ec;
// outcome::result<T> is like expected<T, error_code_extended>
// result<int> can therefore be either an int or an error_code_extended

// Operators & and | work intuitively ...
result<int> a(5);
result<int> b(a & 6);
result<int> c(b | 4);
result<int> d(a & ec);
result<int> e(d & 2);
result<int> f(d | 2);
```

This executes constexpr, so the assembler is `movl #2, %eax`

```
using namespace BOOST_OUTCOME_V1_NAMESPACE;
error_code_extended ec;
// outcome<T> can be a T, an error_code_extended or an exception_ptr
// Operator >> is monadic bind (call callable with current state,
return from callable makes new monad)
outcome<std::string> a("niall");
auto x(a >> [ec](std::string) { return ec; }    // returns

         >> [](error_code_extended) { return
std::make_exception_ptr(5); }
         >> [](std::exception_ptr) { return; }
         >> [](outcome<std::string>::empty_type) { return
std::string("douglas"); });

// What is x?
```

```cpp
using namespace BOOST_OUTCOME_V1_NAMESPACE;
error_code_extended ec;
// outcome<T> can be a T, an error_code_extended or an exception_ptr
// Operator >> is monadic bind (call callable with current state,
return from callable makes new monad)
// Non-C++ monads always copy state, Outcome’s monads can also move,
just ask using a rvalue ref
outcome<std::string> a("niall");
auto z(a >> [](std::string &&v) { return std::move(v); }
         >> [](std::string &&v) { return std::move(v); }
         >> [](std::string &&v) { return std::move(v); }
         >> [](std::string &&v) { return std::move(v); });

assert(z.value() == "niall");
assert(a.value().empty());
```

Thank you

And let the questions begin!

Github: https://github.com/ned14/boost.outcome

Ref docs: https://ned14.github.io/boost.outcome