MAYSTREET

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

Failing Successfully

Reporting and Handling Errors

ERROR:

ERROR: No such file or directory

What Went Wrong?

Actually a success story

- Program acknowledged that it couldn’t proceed
- Gave some vague indication as to why it couldn’t proceed
- Wound itself down properly (didn’t “crash”)

Better than a lot of production programs

- Which motivates this talk

Not All Terminations Are Equal

Ending the program is a very common response to errors

- If something went wrong how can you proceed?

Not all means of terminating a program are desirable

- Program termination through...
  - ...proper structure: **Good**
  - ...`exit` and friends: **Not so much**

What's Wrong With `exit`?

Hidden control flow

- `goto` renders control flow inscrutable intra-function
- `exit` et al. render control flow inscrutable inter-function

Global state

- Commonly understood in terms of **variables**
- More broadly understood as referring to **non-local effects**

Kicking the Can Down the Road

Don’t…

- …couple unrelated decisions
  - Extension of “single-responsibility principle”
  - Coupling decisions hurts ability to reuse code
- …pollute code with non-local concerns and knowledge
  - Errors may result in program termination eventually
  - Termination is not a local responsibility
  - Introducing termination undermines decomposition

Defer decisions until context is available to make them appropriately

What About That Error Message?

Pretty standard error string for `ENOENT`

Message would be useful alongside path of missing file or directory

Somewhere in layers of application decision made to discard that context

- Path was available when calling **open** (for example)
- Context was allowed to expire rather than be preserved for output
- Decision may have been...
  - ...structural (no channel for context)
  - ...functional (context not preserved via available channel)

Integer Parsing

```
int atoi(const char* str);
```

Design is perfect for the success case: Accepts a string, returns an integer

What if the string doesn't contain an integer?

Returns zero, but string could be a valid representation of zero

Effectively assumes string is never non-integer (i.e. that error never occurs)

Ignores any trailing non-integer part

Integer Parsing

```
optional<int> real_atoi(const char* str) noexcept {
  const auto result = std::atoi(str);
  if (result) {
    return result;
  }
  while (*str && std::isspace(static_cast<unsigned char>(*str))) {
    ++str;
  }
  if (*str == '-') { ++str; }
  if (*str == '0') { return 0; }
  return nullopt;
}
```

Integer Parsing

```
long strtol(const char* str, char** str_end, int base);
```

Still returns zero on error

`*str_end` will be set to...
- ...`str` on failure
- ...address of character past last character consumed on success

Can effectively differentiate success and failure

What about...
- ...differentiating different kinds of failure?
- ...determining where the failure occurred?

Integer Parsing

```
struct from_chars_result {
    const char* ptr;
    errc ec;
};
from_chars_result from_chars(const char* first, const char* last,
    T& value, int base = 10);
```

Mechanism to report failure much clearer: `ec`

Can differentiate overflow and non-integer string

`ptr` set to `first` on failure: Still can't determine where failure occurred

Fail Fast, Fail Often

Aforementioned integer parsing functions silently ignore leading whitespace

Callers can easily skip whitespace if they want:

```
std::find_if_not(first, last, [](const char c) noexcept {
    const unsigned char u(c);
    return std::isspace(u);
});
```

What if they consider leading whitespace to be an error?

Ignoring leading whitespace makes a decision on behalf of the user

Error Vocabulary

Standard C-style error reporting uses `int` or an `enum`, for example:

- `errno`
- `CURLcode`

This works in isolation: Cause of failure is transmitted to the caller

What about in composition?

- For example: Function in turn calls POSIX and libcurl functions
- What should be returned to avoid losing context?

`std::error_code`

Combines a “code” (an `int`) with a pointer to a “category”

- Category determines how the code should be interpreted
- Different category with same code interpreted as different error
- Category singleton instance of type derived from `std::error_category`
  - Category identity is assumed to be pointer identity

Error Handling

What if code needs to handle a file not being found?

- `errno`: `ENOENT`
- `CURLcode`: `CURLE_FILE_COULDNT_READ_FILE` et al.

With C-style handling could check for certain well known values

How can this be accomplished with `std::error_code`?

- Number of possible errors theoretically unbounded

`std::error_condition`

Same basic structure as `std::error_code`

Intended to encapsulate root cause which can be consumed programmatically

Can be compared to `std::error_code`

- `std::error_category::equivalent` used for comparison
- Doesn't necessarily model an "equality relation"
  - `std::error_code` can be equal to many `std::error_condition`
  - And vice versa

```cpp
enum class error { success = 0, bad_whole, no_decimal, bad_decimal };

std::error_code make_error_code(error e) noexcept {
  static const struct : std::error_category {
    virtual const char* name() const noexcept override {
      return "Decimal Parser";
    }
    virtual std::string message(int code) const override {
      switch (static_cast<error>(code)) {
      case error::success:
        return "Success";
      // ...
      default:
        break;
      }
      return "Unknown";
    }
    virtual std::error_condition default_error_condition(int code) const noexcept
      override
    {
      if (code) return std::errc::invalid_argument;
      return {};
    }
```

```
    //  These are the default implementations inherited from std::error_category
    virtual bool equivalent(int code, const std::error_condition& condition) const
      noexcept override
    {
      return default_error_condition(code) == condition;
    }
    virtual bool equivalent(const std::error_code& code, int condition) const
      noexcept override
    {
      return (*this == code.category()) && (code.value() == condition);
    }
  } category;
  return std::error_code(static_cast<int>(e), category);
}

namespace std {
template<>
struct is_error_code_enum<error> : true_type {};
}
```

`std::system_error`

Exception type which wraps a `std::error_code`

Can derive and provide custom `what` to bundle additional context

- Frames up the stack can catch and handle `std::system_error`
- Alternately can catch `std::exception` and print `what`

Use of this type supposes that we should be throwing an exception

Exceptions

Common to say that exceptions are for “exceptional” situations
    Deeming something “exceptional” makes a decision on behalf of user

Exceptions simplify…
    …error reporting: Just throw
    …context propagation: Add to exception type, provide custom `what`

Exceptions complicate…
    …error handling: What to catch?
    …code analysis: What can fail and how?

The higher level the building block the more appropriate exceptions become

Tag could not be parsed as an integer

```
8=FIX.4.2\x019=00238\x0135=D\x0134=160\x0149=P98004N\x015a=004\x0152=2
                                                                 ^
Tag could not be parsed as an integer
```

```
struct fix_message_reader {
  // ...
  fix_message* next(std::error_code& ec);



};
```

```
struct fix_message_reader {
  // ...
  fix_message* next(std::error_code& ec);
  std::string format_last_error() const;



};
```

```
struct fix_message_reader {
  // ...
  fix_message* next(std::error_code& ec);
  std::string format_last_error() const;
  const std::byte* last() const noexcept;
  const std::byte* last_begin() const noexcept;
  const std::byte* last_end() const noexcept;
  const std::byte* begin() const noexcept;
  const std::byte* end() const noexcept;
};
```

```
struct standard_fix_client : /* ... */ {
  // ...
  std::string format_last_error() const;



};
```

```
struct standard_fix_client {
  // ...
  std::string format_last_error() const;
  enum class error_source {
    parsable,
    verify,
    parse_fix,
    parse_unknown,
    stop,
    other
  };
  error_source last_error_source() const noexcept;
  fix_message_reader& message_reader() noexcept;
  const fix_message_reader& message_reader() const noexcept;
  // ...
};
```

Multi-Threading

Reporting errors via returned value supposes there's a returned value

Non-trivial programs tend to have multiple threads

- "Returned value" doesn't make sense in this context
- Requirement to handle errors still exists
  - Need to "gather" errors from all threads
  - Also need to be able to stop if one thread encounters an error

```
class thread_pool {
  struct state : ::asio::io_context {
    std::thread thread;
  };
  using states_type = std::list<state>;
  states_type states_;
  mutable std::mutex m_;
  std::exception_ptr ex_;
public:
  explicit thread_pool(unsigned threads);
  void run();
  void stop(std::exception_ptr ex = std::exception_ptr()) noexcept;
  using iterator = states_type::iterator;
  iterator begin() noexcept;
  iterator end() noexcept;
};
```

```
void thread_pool::run() {
  const auto run = [&](auto&& ctx) noexcept {
    try {
      ctx.run();
    } catch (...) {
      stop(std::current_exception());
    }
  };
  {
    auto begin = std::next(states_.begin(), 1);
    const auto g = make_scope_exit([&]() noexcept {
      for (auto iter = std::next(states_.begin(), 1); iter != begin; ++iter) {
        iter->stop();
        iter->thread.join();
      }
    });
    for (const auto end = states_.end(); begin != end; ++begin) {
      begin->thread = std::thread([&, begin]() noexcept { run(*begin); });
    }
    run(states_.front());
  }
  const std::lock_guard g(m_);
  if (ex_) std::rethrow_exception(std::move(ex_));
}
```

```
10.244.0.33:41534 => 0.0.0.0:11653 disconnected due to failure reading
from socket: End of file
```

Whose Error?

Whether something is an error depends on...

- ...level of abstraction
  - `read` does not consider end of file to be an error
  - Attempting to fill a buffer we may treat it as an error
  - Managing connections may not consider it an error: Stream is done
- ...purpose
  - Invalid XML is an error when parsing XML
  - Not an error when trying to heuristically determine if a file is XML

Succeed, Fail, Who Cares?

What does it mean for a TCP connection to “succeed?”

- Useful distinction to a client, but for a server?
- Success might mean “goodbye” message received or graceful shutdown
  - Does that really matter?
  - Connection is still gone
  - Doesn’t affect overall server
  - Failure and success handled in essentially the same manner

```
struct processor_manager_settings {
  // ...
};

struct processor_manager {
  explicit processor_manager(
    const processor_manager_settings& settings);
  void add_device(device& d);
  void add_feed(feed& f);
  void start();
  void stop() noexcept;

};
```

```
struct processor_manager_callback;

struct processor_manager_settings {
  // ...
};

struct processor_manager {
  explicit processor_manager(
    const processor_manager_settings& settings);
  void add_device(device& d);
  void add_feed(feed& f);
  void start();
  void stop() noexcept;
  void subscribe(processor_manager_callback& callback);
};
```

```
struct device_processor_begin {
  device_processor& processor;
};
struct packet_processor_begin {
  packet_processor& processor;
};
struct device_processor_end : device_processor_begin {



};
struct packet_processor_end : packet_processor_begin {



};
struct processor_manager_callback {
  virtual void on(const device_processor_begin& e) = 0;
  virtual void on(const packet_processor_begin& e) = 0;
  virtual void on(const device_processor_end& e) = 0;
  virtual void on(const packet_processor_end& e) = 0;
};
```

```
struct device_processor_begin {
  device_processor& processor;
};
struct packet_processor_begin {
  packet_processor& processor;
};
struct device_processor_end : device_processor_begin {
  std::error_code ec;
  std::exception_ptr ex;
  device* which;
};
struct packet_processor_end : packet_processor_begin {
  std::exception_ptr ex;
  session* which;
};
struct processor_manager_callback {
  virtual void on(const device_processor_begin& e) = 0;
  virtual void on(const packet_processor_begin& e) = 0;
  virtual void on(const device_processor_end& e) = 0;
  virtual void on(const packet_processor_end& e) = 0;
};
```

```
struct eof_processor_manager_callback :
  processor_manager_callback
{
  virtual void on(const device_processor_begin& e) override;
  virtual void on(const packet_processor_begin& e) override;
  virtual void on(const device_processor_end& e) override;
  virtual void on(const packet_processor_end& e) override;
  enum class processor { packet, device };
  processor source() const noexcept;
  const std::string& name() const noexcept;
  void wait() const noexcept;
  bool eof() const noexcept;
  void maybe_throw() const;
};
```

Warnings & Logging

Forms of out of band communication
- Succeed but also warn
- Fail but also log

Logging can be used in short term to compensate for lack of error reporting
- Short term because logging isn’t always appropriate

Warnings should have a bona fide channel

Logging shouldn’t be coupled into components
- Emit events
- Separate component consumes events and writes to log

Summary

Don’t…

…assume failure conditions won’t happen

…unnecessarily make decisions on behalf of your user

…throw away potentially useful context

ErrorInvalidRange

```
Domain: libs3 code domain
Description: ErrorInvalidRange
File: ../src/apps/cme/data_conn/product_info_query.cpp
Line: 61
Built: Sep 17 2021 23:08:39
Revision: 1035-f685c515fa6c89fe25c27e9fc3fe89d88735f83f
Database: /db
Bellport Revision: 10742-35ad327965de328e7bf3e6823102c46827516b54
Session: 387
DXF Type: Symbols
MIC: GLBX
Date: 2021-08-06
Identifier: b6 4e 11 49 9d 9e c8 b8 00 00 00 01 83 04 06 00
```

Questions?

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca

MAYSTREET