@andrewcmorrow #nyccpp

NYC C++ Jan 13 Meetup: error_or

Andrew Morrow -- acm@10gen.com

Member Technical Staff, 10gen (the MongoDB company)

Caveats

- I’m still learning about this stuff
- I’m almost surely wrong about some things here.
- If you think I’m wrong about something, please tell me:
  - I don’t want to spread misinformation
  - I learn something too

Why not throw?

- Because you work at Google.
  - Because you previously worked at Google.
  - Because your team lead used to work at Google.
- Because the framework you are using is not exception safe.
- Because you are on a constrained environment and your runtime doesn't offer it.
- Because writing exception safe code is really hard.

Error handling without exceptions

- Return an error code and use out parameters.
  - Loses return slot, awkward to call.
  - https://gist.github.com/4589211
- Return a value, and have a (thread local?) getLastError function.
  - You must return a “bad” value, easy to forget to check error.
  - https://gist.github.com/4589420
- Return std::pair<error_type, value_type>
  - Uses extra space; how do I know which is valid?
  - https://gist.github.com/4589450

Pretty OK: return boost::variant<T, E>

- Like returning std::pair, but only contains either T value or E value.
  - Requires dependency on boost.
  - boost::variant is complicated (non-empty guarantee).
  - We can do better by using C++11 and rolling our own type.

error_or<E, V>

- Code: https://github.com/acmorrow/error_or
  - Requires C++11
    - Final, new style unions, move semantics, noexcept, std::error_code, etc.
  - Logical successor to boost::variant approach.
  - Stores either an E value to represent and error, or a V value to represent a returned value.
  - Intended to be pure "type calculus", it should compile away to operations on the underlying types, minimal perf hit.
  - Designed for use with std::error_code or std::error_condition as the 'E' type, but would work with others.

Note on code links

- The links here are to a specific revision to preserve line numbers:
  - https://github.com/acmorrow/error_or/tree/f7c0092c02b153f73e8dfb806f0a3ec9018a2593
  - Subsequent revisions may have slightly different line numbers but should be in the same relative position in the file.
  - Code links in this presentation will list the file and line if you want to view the code on github during the talk.
    - Head revision may slightly differ

Declaration

- error_or.hpp: L28-L33
  - First template argument is the error type that is returned to indicate failure:
    - Just about any type will do, but intended to be std::error_code or std::error_condition.
  - Second argument is the value returned on success.
  - Class is final so we know there are no subclasses, no need for a virtual dtor, etc. Almost call calls should inline away.
  - Add STL style typedefs for T (value_type) and E (error_type) so that metaprogramming can be done with this class.

Review: Placement new

- ‘new expression’ normally does two things:
  - Obtain a region of appropriately sized and aligned memory
  - Invokes the constructor
- Placement new:
  - Does not allocate dynamic memory.
  - Takes an argument, which is an address.
  - Returns the address it is passed.
  - Invokes the constructor at that returned address.
- Placement new allows you to construct an object into a given region of existing memory.

Review: Explicit call to destructor

- Complement to placement new:
  - Invokes the destructor directly.
    - t->T::~T();
    - t.~T();
  - Does not invoke dynamic memory allocator.
- Allows us to 'tear down' an object that exists at a certain memory location, leaving that memory region uninitialized (and therefore able to host a new object in the future)

Members

- error_or.hpp: L176-L185
  - 'val_': A union of non-trivial class types (a.k.a 'unrestricted unions')
    - Not permitted in C++03, OK in C++11
    - Requires that we use placement new and explicit calls to the dtor to initialize and destroy members.
    - The empty val ctor/dtor inhibit the compiler from generating members for us.
  - 'ok_': default initialized in the class body, tells us which field of the union is populated ('discriminator').

Lifecycle

- C++03: Rule of 3
  - dtor, copy ctor, copy assign operator: nontrivial existence of any suggests need for all.
    - For templates, maybe include converting copy ctor and converting move assign operator.
- C++11: Rule of 5?
  - All Rule of 3 members plus new special functions: move constructor, move assign operator
    - For templates, maybe include converting move ctor and converting move assign.

C++11: Rule of Zero when possible

- Don’t write lifecycle members unless you need to
  - If all of your class members have good semantics, you may not need to write any special members yourself anyway.
    - Use containers, C++11 smart pointers, RAII, etc.
    - Use ‘=default’ if you want to be explicit.
    - Use ‘=delete’ to disable behavior.
- For error_or though we need to be explicit since we manage memory directly (the union slots).

C++11 noexcept

- Adding 'noexcept' specifier to a function states that it will not throw.
  - Well, actually, that if it does throw, to call std::terminate.
  - Much simpler codegen than honoring exception specifiers.
    - Exception specifiers are deprecated in C++11.
- For a template it may depend on properties of template type parameters:
  - noexcept operator: noexcept(<expression>)
  - std::is_nothrow_[action]able

Default Constructor

- error_or.hpp: L45-L47
  - Only works if value_type has a default constructor
  - We use placement new to initialize the appropriate (in this case ‘value’ obviously) member of the union.
  - ‘ok_’ is true because of the in class body initializer.
  - Is noexcept, as long as value_type’s default constructor is noexcept.
  - The remaining constructors use std::move and move semantics…

What does std::move do anyway?

- It doesn't 'move' anything, it **enables** moves:
  - "obtains an rvalue reference to its argument"
  - The return value of std::move can bind to an argument that takes an rvalue reference.
  - void some_func(Foo&& f) {
    - Foo wrong(f); // NO: calls copy ctor!
    - Foo right(std::move(f)); // YES: calls move ctor.
      - 'f' is 'dead' at this point.
  - When in doubt: && => use std::move
    - Unless you need std::forward (not covered today)
      - See Meyer's talk on 'Universal References' for details

Value and Error Constructors

- error_or.hpp: L49-L57
  - If the argument is an error_type, then we set ‘ok_’ to false, and placement new into the val_.error field.
  - If the argument is a value_type, then we placement new into the val._value field.
  - We take the arguments by value (but may be zero copy!).
  - We enable moving from the copied arguments by using std::move.
    - The copied arg is private, so it’s OK if we destroy it.

Destructor

- error_or.hpp: L135-L141
  - We switch on ‘ok_’ to determine which union slot is currently populated, and then make an explicit call to the destructor on the value in the indicated slot.
  - Good ‘noexcept’ example: this function is noexcept if the dtor’s for both error_type and value_type are noexcept.
    - We can use std::is_nothrow_destructible to find out.
    - Use the result of the above in a noexcept expr.

Copy Constructor

- error_or.hpp: L64-L71
  - Argument is passed by const&, as you would expect.
  - Introspect ‘other.ok_’ to determine which field in other is valid, and use placement new to initialize the same field in our union with a copy of the appropriate field from other.
  - this->ok_ gets the same value as other.ok_
  - Why can’t we take these arguments by value then move, like we did before?

Move Constructor

- error_or.hpp: L73-L82
  - Much like copy ctor, but we want to move from the rvalue reference.
  - Use of std::move_if_noexcept may be too conservative, using std::move here might be fine.
    - std::move_if_noecept works like std::move, but if the type's move constructor is not noexcept, returns 'const T&', inhibiting moveage.

What about assignment?

- Construction is “easy”: copy or move the provided argument into the correct union slot with placement new.
- Assignment is hard: the target error_or already has state that we must tear down.
  - We would like strong exception safety
  - ‘cis’ assignments (error valued to error valued, value valued to value valued) are relatively straightforward.
  - ‘trans’ assignments are hard
    - What if one move works, but the other throws?
    - If we catch, and move back, and *that* throws?

Use ‘copy and swap’ for assign

- Copy and swap is the gold standard for exception safe copy assignment operators:
  - Make a copy of the thing you are assigning from
    - It’s ok, you will have two at the end anyway.
  - Swap your internals with the copy (this is cheap).
  - Copy goes out of scope (destruction was inevitable).
  - Relies on exception safety of swap.
- We have merely pushed the problem to swap.
  - Solution: Use SFINAE to disable swap, and therefore assign, if we can’t build a non-throwing swap over E and T.

Copy Assignment Operators

- error_or.hpp: L114-L127
  - Take arguments by value to achieve 'copy' step.
    - Again, may in fact not require any copying.
  - Invoke our move constructor to steal the copy from the argument.
  - Swap our internals with the newly constructed error_or.
  - All is well, as long as swap won't throw…

Exception safe swap?

- error_or.hpp: L187-L210
  - Maybe. We use std::enable_if to SFINAE this swap in or out depending on a class constant ‘is_nothrow_swappable’
  - Requirements for error_or::is_nothrow_swappable:
    - error_or.hpp: L36-L42

Review: ADL Swap

- Old advice: specialize std::swap in namespace std for your type
  - Ugly, and it doesn't work for templates anyway.
- Better way: leverage 'argument dependent lookup' to find swap in your own namespace
  - Clean, and works for templates
  - Just write your in-namespace swap as a two argument free function.
- Relies on callers always invoking swap with no qualifier.
  - Say 'using std::swap' first so you fall back to std variant.

Exception safe swap (part 2)

- For ‘cis’ swap, we can swap with no exception if value_type and error_type can swap with no exception.
  - There is no std::is_[nothrow_]swappable!
    - This seems like an oversight in the standard?
  - We will roll our own:
    - is_nothrow_swappable.hpp
    - Pretty complex. Is it correct?

detail::is_[nothrow_]swappable

- Enable ADL swaps by ‘using std::swap’ in a private namespace
- SFINAE on decltype(swap(…, …)) to check if swappable
  - Build an integral constant from the result
- Use noexcept(swap(…,…)) to check for nothrow swap, but only if the underlying type is swappable.
- Cobbled together from various sources,

Exception safe swap (part 3)

- For ‘trans’ swap, we need noexcept move and dtor
  - The standard gives us these
    - std::is_nothrow_move_constuctible
    - std::is_nothrow_destructible
- With our shiny new is_nothrow_swappable and the above metafunctions, we can determine if our swap is nothrow.
  - We only emit the swap if it is nothrow.
  - Assignment operators won’t be available otherwise.

Adding member and ADL swap

- error_or.hpp: L106-L112
  - Our member swap delegates to our SFINAE'd sfinae_swap.
  - Our ADL swap delegates to member swap.
  - Both members expose noexcept status.
    - detail::is_swappable works on error_or<T>.
    - detail::is_nothrow_swappable works on error_or<T>.

Move and noexcept are friends

- If you must write a move special member despite the rule of zero, and you can prove that it doesn't throw, then you should mark it with noexcept.
  - Allows templates instantiated on your type to determine that moves can be used safely.
  - Allows templates instantiated on your type to propagate noexcept on their own special member functions.
  - Enables non-throwing std::swap or ADL swap via std::move.
    - Which enables exception safe assignment via copy and swap.

Accessors: Some easy stuff...

- error_or.hpp: L143-L174
  - ‘ok()’ tells us whether we have a value or an error.
  - operator bool() for use in conditionals
  - value and error accessors return the identified object.
  - ‘release’ functions to move values or errors out.

Integration with <system_error>

- error_or.hpp: L243-L247
  - Use C++11 'template typedefs' via 'using' to define nice names for error_or<std::error_[condition|code], T>
  - No need to roll your own 'error type', just make a new category and use what the standard gives you.
  - These types are noexcept.

Integration with std::unique_ptr

- error_or.hpp: L249-L253
  - std::unique_ptr makes a great return value for functions that create a new object, and has noexcept move semantics.
  - Combining it with error_code_or gives you the following nice semantics:
    - The function returns to me an owned pointer to a newly created object, or an error explaining why it was not constructed.
  - Example:
    - error_code_or_unique<socket> connect_to(std::string host);

Fun and Games

- Convert a function that returns T or throws a std::system_error into a function that returns error_code_or<T>:
  - converters.hpp: L25-L45
  - I suspect there is a better way to declare this, but I haven't derived it yet.
- Convert a function that returns error_code_or<T> into one that returns T or throws:
  - converters.hpp: L47-L65
  - Again, there might be a better way to do this.

Fun and Games: In Progress

- I'd like this to work:

```
error_code_or_unique<socket> connect(std::string host) {...}
error_code_or_unique<request> socket::send(std::string message)
{...}

auto response = connect("www.10gen.com").switch(
  [](std::unique_ptr<socket>&& sock) ->
error_code_or_unique<response> {
    return sock->send("GET /index.html http/1.1").switch(
      [sock](std::unique_ptr<request>&& request) {
        return request->await_response(std::move(sock));
      },
      [](std::error_code&& error) ->
error_code_or_unique<response>{
        std::cerr << "Failed to send: " << error << std::endl;
        return error;
      }
    )
  },
  [](std::error_code&& error) -> error_code_or_unique<response> {
    std::cerr << "Failed to connect: " << error << std::endl;
    return error;
  }
);
```

Examples and Demo

Thanks!