D0779R0: Proposing `operator try()`

Document #: D0779R0 draft 2
Date: 2017-10-12
Project: Programming Language C++
Evolution Working Group
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

Something which would be useful to the LEWG Expected proposal [P0323], the C++ Monadic Interface proposal [P0650] and the proposed Boost.Outcome library `https://ned14.github.io/outcome/` would be if we could customise the `try` operator in a similar way to how Swift[1] and Rust[2] implement `try`. This allows one to more succinctly implement a *lightweight failure handling alternative to exception throws* without typing so much tedious boilerplate all the time.

Example in code:

```cpp
// Without operator try
template<class T> using expected =
  std::expected<T, std::error_code>;

expected<int> get_int() noexcept;

expected<float> get_float() noexcept
{
  expected<int> _int = get_int();

  // If get_int() failed, propagate the error
  if(!_int)
    return unexpected(_int.error());
  float ret = (float) *_int;

  // If the float cannot wholly represent
  // the int, return an error
  if((int) ret != *_int)
    return unexpected(std::errc::
        result_out_of_range);

  // Otherwise return success
  return ret;
}
```

```cpp
// With operator try
template<class T> using expected =
  std::expected<T, std::error_code>;

expected<int> get_int() noexcept;

expected<float> get_float() noexcept
{
  int _int = try get_int();




  float ret = (float) _int;

  // If the float cannot wholly represent
  // the int, return an error
  if((int) ret != _int)
    return unexpected(std::errc::
        result_out_of_range);

  // Otherwise return success
  return ret;
}
```

In other words, we just want to 'inject' some type-specific boilerplate into the calling scope in a

[1] The `try` keyword in Swift (`https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html`).

[2] The `try!` macro in Rust (`https://doc.rust-lang.org/std/macro.try.html`).

similar way to how the [N4680] Coroutines TS implements `co_await`.

1 Motivation

1.1 Frequency of use

Those who have not programmed in Rust nor Swift, and are not practised in writing code which uses Expected|Outcome extensively, are not aware how frequently one performs the `try` operation.

With C++ exception handling, the points at which control flow can change are *invisible*. This is not the case with Expected|Outcome code where the programmer must explicitly annotate each potential control flow change point with either explicit `if` logic, or a `try`. For obvious reasons, these rapidly proliferate and become tedious to constantly write, so programmers will seek shortcuts to avoid constantly writing the same boilerplate again and again.

Due to such frequency of use, without language support for `try`, one inevitably would use a C macro expanding into a GCC/clang language extension called 'statement expressions'[3]. Here is Outcome's implementation:

```
#define OUTCOME_TRYX(m) \
  ({ \
    auto &&res = (m); \
    if(!res.has_value()) \
      return OUTCOME_V2_NAMESPACE::try_operation_return_as(std::forward<decltype(res)>(res)); \
    std::forward<decltype(res)>(res).value(); \
})
```

The use of C macros is not ideal. The use of a non-standard language extension is worse again. This has bothered people enough to seek workarounds by misusing the C++ language.

[3] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

1.2 Failure to standardise this means people will abuse `co_await` to achieve the same thing

In September 2017, Facebook Folly's Optional gained the ability to be awaited upon with `co_await`[4]. This is not a true coroutine await, rather it's an abuse of awaiting to inject boilerplate due to the C++ language's inability to otherwise do this. Quoting from a CppCon 2017 talk called 'Coroutines: What can't they do?' by Toby Allsopp[5]:

```
optional<vector<double>> parse_vector(istream& s) {
  optional<int> n = parse_int(s);
  if(!n) return ();
  vector<double> result;
  for(int i = 0; i < *n; ++i) {
    optional<double> x = parse_double(s);
    if(!x) return {};
    result.push_back(*x);
  }
  return result;
}
```

```
optional<vector<double>> parse_vector(istream& s) {
  int n = co_await parse_int(s);

  vector<double> result;
  for(int i = 0; i < *n; ++i) {


    result.push_back(co_await parse_double(s));
  }
  co_return result;
}
```

I find this misuse very troubling for all the obvious reasons, and I hope so do you as well. This needs to be nipped in the bud before it goes septic and starts appearing across the C++ ecosystem.

2 Solutions

I will propose two potential solutions to the problem of injecting the necessary type-specific boilerplate for an `operator try`: (i) a narrow proposal and (ii) a wide proposal.

2.1 Implement `operator try` just like `operator co_await`:

```
template <class T, class E>
constexpr auto operator try(std::expected<T, E> v) noexcept
{
  struct tryer
  {
    std::expected<T, E> v;

    constexpr bool try_return_immediately() const noexcept { return !v.has_value(); }
    constexpr auto try_return_value() { return std::move(v).error(); }
    constexpr auto try_value() { return std::move(v).value(); }
  };
  return tryer{ std::move(v) };
}
```

[4] https://github.com/facebook/folly/blob/master/folly/Optional.h
[5] https://www.youtube.com/watch?v=mlP1MKP8d_Q, about 30 mins in.

```
// Introductory example expanded
template<class T> using expected = std::expected<T, std::error_code>;

expected<int> get_int() noexcept;

expected<float> get_float() noexcept
{
  int _int = try get_int(); /*  --> auto __unique = operator try(get_int());
                                    if(__unique.try_return_immediately())
                                      return __unique.try_return_value();
                             _int = __unique.try_value();
  */

  float ret = (float) _int;

  // If the float cannot wholly represent
  // the int, return an error
  if((int) ret != _int)
    return unexpected(std::errc::result_out_of_range);

  // Otherwise return success
  return ret;
}
```

If implementing `co_await` this way it is is uncontroversial, then I guess so is the above. It solves the direct problem at hand quickly and simply.

But can we solve this whole class of injecting boilerplate problems in one fell swoop, now and forever?

2.2 Implement `operator try` by adding native C++ macros to the language

This section likely could form a paper of its own ☺. If you like the idea, please do feel free to submit a P-paper proposing it. I'm no language person, I'm the wrong one to propose it seriously.

Operator try is hitting the exact same problem as the Coroutines TS ran into when implementing `co_await`: **boilerplate injection**. C++'s current method of injecting boilerplate is the C preprocessor, and it is non-ideal for a long list of reasons which is why the Coroutines TS adopted its solution which looks exactly like our solution in the preceding section.

But what if C++ had a language feature for injecting boilerplate? Rust has a feature like this which it calls 'macros'[6]. These are normal functions, but their contents (tokens) are injected into the point of use as-is.

Could we perhaps implement the same thing in C++? Well we can't use the bang token '!' because `return!(v);` might mean 'inject contents of the return! macro' or it might mean 'return logical NOT of v', and the same rationale applies to all C++ operator tokens except possibly for '?' and ':'.

[6] https://rustbyexample.com/macros.html

But it turns out that the '#' token is available to us: the C preprocessor must emit a '#' token if it is not the first non-whitespace token in a line and is not inside a parameterised macro definition. Moreover, GCC, clang and MSVC all error out about stray '#' tokens if they leak into the preprocessor output. Therefore, no valid code is out there using the '#' token in identifier names, and is available to us for this use case.

So let's turn this idea into example code:

```
/* This function's identifier ends with a # token, and thus
is to be treated as a collection of unprocessed tokens by
the compiler. You can template the arguments and contents
of course. The identifier is otherwise like a normal free function,
they are namespaced, participate in ADL etc.

These look a little like the GCC/clang extension
https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html
but they really are a bunch of unprocessed tokens
injected into the use site, except for the final
expression which is the "output" of the macro.
*/
template<class T> void return#(T v)
{
  if(v > 0)
    return v;
  -1;  // the output to the call site
}

int function(int a)
{
  // You must call including the '#' so the programmer
  // and compiler knows that this injects tokens right here
  int v = return#(a);  /* if(a > 0) return a; v = -1; */
  return v;
}
```

2.2.1 Implementing `co_await` using these native C++ macros

Let's see how we might implement `co_await` using these.

`auto ret = co_await awaitable_expr;` is effectively this pseudo-code:

```
auto __unique = awaitable_expr;
// Is the awaitable in __unique not ready?
while(!__unique.await_ready())
{
  // this_coroutine_handle() returns the coroutine_handle<> for this coroutine
  // Tell the awaitable we are about to suspend
  __unique.await_suspend(this_coroutine_handle());
  // Suspend this coroutine
  __builtin_coroutine_suspend();
  // When it returns here we are resumed
}
// Ask the awaitable for the value to emit from the co_await operator
auto ret = __unique.await_resume();
```

So instead of the complex `operator co_await` currently proposed in the Coroutines TS, we get instead this:

```
void co_await#(auto awaitable_expr)
{
  // Is the awaitable_expr not ready?
  while(!awaitable_expr.await_ready())
  {
    // this_coroutine_handle() returns the coroutine_handle<> for this coroutine
    // Tell the awaitable we are about to suspend
    awaitable_expr.await_suspend(this_coroutine_handle());
    // Suspend this coroutine
    __builtin_coroutine_suspend();
    // When it returns here we are resumed
  }
  // Ask the awaitable for the value to emit from the co_await operator
  awaitable_expr.await_resume();
}
```

And voilá, `co_await#()` nicely replaces `co_await` in a much more flexible, **entirely library defined**, fashion which means that the original name of `await#` can be used instead, along with `yield#` and `return#` instead of the ugly `co_return#`[7]. No core C++ language changes with new keywords needed.

Let's end with implementing `try` for Expected using this new mechanism:

```
template <class T, class E>
void try#(std::expected<T, E> v)
{
  // If there is an error, propagate that error immediately
  if(!v.has_value())
    return std::move(v).error();
  // Otherwise the output of this macro is the value.
  std::move(v).value();
}


// Introductory example expanded
template<class T> using expected = std::expected<T, std::error_code>;

expected<int> get_int() noexcept;

expected<float> get_float() noexcept
{
  int _int = try#(get_int());
  float ret = (float) _int;

  // If the float cannot wholly represent
  // the int, return an error
  if((int) ret != _int)
    return unexpected(std::errc::result_out_of_range);

```

[7] Me personally I'd have coroutines declare a `using namespace std::coroutines;` at the top of each coroutine function. This would tell the compiler that this is (a) potentially a coroutine, watch out for suspension points and (b) bring in the macro definitions for use without namespace prefixing.

```
  // Otherwise return success
  return ret;
}
```

This isn't *quite* as nice as the earlier `operator try`, but it sure beats `OUTCOME_TRYX(expr)`.

3 Acknowledgements

- Vicente J. Botet Escribá for his extensive commentary on earlier drafts of this paper.
- std-proposals for helping me work through the 'native C++ macros' idea.
- Michael Park for making available this LaTeX template at https://github.com/mpark/wg21/.

4 References

[P0650] Vicente J. Botet Escribá,
C++ Monadic interface
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0650r0.pdf

[P0323] Vicente J. Botet Escribá,
A proposal to add a utility class to represent expected object (Revision 4)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r2.pdf

[P0262] Lawrence Crowl, Chris Mysen,
A Class for Status and Optional Value
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0262r0.html

[N4680] Gor Nishanov,
C++ Extensions for Coroutines TS
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf