P0779R0: Proposing `operator try()` (with added native C++ macro functions!)

Document #: P0779R0
Date: 2017-10-15
Project: Programming Language C++ Evolution Working Group
Reply-to: Niall Douglas <s_sourceforge@nedprod.com>

Something which would be useful to the 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 saves having to type so much tedious boilerplate when writing code with Expected 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;
}
```

[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`).

In other words, we just want to 'inject' some type-specific boilerplate into the calling scope in a 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 would probably reach for a C macro expanding into a GCC/clang language extension called 'statement expressions'[3]. Here is one potential implementation[4]:

```
#define TRYX(m) \
  ({ \
    auto res = (m); \
    if(!res.has_value()) \
      return unexpected(res.error()); \
    res.value(); \
})
```

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

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

[4] Yes, it is incorrect because it doesn't handle non-moveable and non-copyable type `T`. It has been simplified for brevity of presentation here.

1.2 Failure to standardise this means people may 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`[5]. 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 at the language level. Quoting this code example from a CppCon 2017 talk called 'Coroutines: What can't they do?' by Toby Allsopp[6]:

```
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;
}
```

There are some obvious big problems with this type of abuse of `co_await`. As soon as `co_await` appears in a function, it becomes *coroutinised* and is no longer a normal function. So, you must return via `co_return`, the function's return type must be an awaitable, and the function returning infrastructure must become owned and managed by coroutines. Furthermore, you cannot await on something different to the coroutine return type e.g. a `future<std::string>` when the coroutine returns a `future< expected<int> >`.

For these reasons and more, I find this misuse of `co_await` troubling, 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 due to lack of a better alternative.

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
```

[5] `https://github.com/facebook/folly/blob/master/folly/Optional.h`

[6] `https://www.youtube.com/watch?v=mlP1MKP8d_Q`, about 30 mins in.

```cpp
  {
    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) };
}


// 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` using the same design pattern as the above 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++ macro functions 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 dedicated language feature for injecting boilerplate? Rust has a feature like this which it calls 'macros'[7]. These are normal functions, but their contents are injected into the scope of the point of invocation. Indeed, Rust's `try!` operation is implemented using just such a macro (one can distinguish macro functions in Rust by the bang token '!' at the end of the function name).

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 function' or it might mean 'return logical NOT of v', and the same rationale applies to all C++ operator tokens except possibly for '?' and ':'.

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.

[One issue I don't know if it is important or not is that the C preprocessor when tokenising the input will treat identifiers as not being able to have '#' in them, so `#define foo# boo` *will generate a preprocessor error. There are already quite a few instances where C and C++ tokenise an input differently, but how important it is that the C preprocessor can token swap the names of these proposed C++ macro functions I don't know]*

So what would one of these native C++ macro functions look like?

```
// Macro functions look just like a free function except for the # at the
// end of the function name. Note that the # counts as part of the identifier,
// so return# does not collide with the return keyword.
template<class T>
inline int return#(T v)
{
  if(v > 0)
    return -> v;  // control flow keyword + '->' means it affects the
                  // calling function, not this macro function
  if(v < 0)
    break ->;     // Also this break is executed in the calling function
                  // If break isn't valid at the point of use
                  // in the calling function, it will not compile

  // We can inject variable declarations into the calling function
  // with typename + '->'. This is useful for RAII triggered cleanup
  // i.e. these get destructed when the scope of the call point exits.
  // Note that the actual name of the variable injected will
  // be some very unique identifier which cannot collide with any
  // other variable, including those injected by other macro functions
  int -> a = 5;

  // Otherwise this function macro has a local scope, and code
  // executed here remains here
  size_t n = 0;
  for(; n < 5; n++)
  {
```

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

```
    // We can also refer to variables previously injected into the
    // caller’s scope by this macro function like this.
    // This lets one keep state across invocations of the macro function
    (-> a) ++;
  }

  // This returns a value from this function macro to the caller
  // If you wrote return -> a, that would be a compile error
  // as there is no variable called a in this scope.
  return (-> a);
}
```

To summarise, macro functions are like normal free functions except that they are attached to the scope of their calling non-macro function. They can inject control flow into their calling function by placing the ‘->’ token after a C++ keyword, instantiate variables into a per-macro-function scope with the same lifetime as the calling non-macro-function’s scope, and retrieve/modify variables in that per-macro-function scope previously instanced.

And in action:

```
static int values[];  // some array somewhere
int foo(int n)
{
  int acc = 0;
  for(;;)
  {
    // Macro functions are invoked just like normal functions,
    // no token pasting. So n will be incremented exactly once.
    acc += return#(values[n++]);
  }
}
```

My personal, not fully thought through, recommendations for these native C++ macro functions:

- Macro functions are namespaced, overloadable and participate in ADL etc just like ordinary free functions. This lets you write `foo#(something)` and ADL will find a most appropriate `foo#` macro function overload.
- Macro functions must always be fully defined at the point of invocation in the current translation unit (Modules notwithstanding). They cannot be `extern`.
- Macro functions must always be `inline`, though as they can’t have linkage anyway, this is a moot point.
- You cannot take the address of a macro function. It does not exist as a free standing function.
- Their identifiers do however identify a unique type each, and thus can be passed around wherever a template type would be allowed. You can thus supply them to templates e.g.

```
template<template<class> class ReturnMacro#> int foo(int n)
{
  int acc = 0;
  for(;;)
  {
    // Macro functions are invoked just like normal functions,
    // no token pasting. So n will be incremented exactly once.
    int ret = ReturnMacro#<int>(values[n++]);

    // int a was injected here by the macro function
    acc += a;
  }
}
...
foo<return#>(5);
```

Note how the template parameter's name must also end with a '#', otherwise it wouldn't compile.

- Macro functions can of course call other macro functions. The `->` injection of control flow or variables always happens to the non-macro-function caller, not to any macro function caller.
- If a set of macro functions need to share the same injected state, simply use an injected state accessor macro function for retrieving a lvalue reference to it as the injected variables are permuted by the macro function's fully qualified identifier, so the same macro function always sees the same injected variable namespace and no other macro function sees another's.

2.2.1 Implementing `co_await` using these native C++ macro functions

As a test to see if they are fit for purpose, 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?
if(!__unique.await_ready())
{
  /* Coroutinised functions are compiled twice into non-detached
  and detached editions. We enter the non-detached edition, if it ever
  suspends then resumption is to the detached edition. So imagine
  that this fictitious intrinsic suspends execution, resuming
  in the detached edition of this function */
  coroutine_handle<promise_type> coro = __builtin_coro_suspend(resume_detached);

  // Tell the awaitable we have suspended
  __unique.await_suspend(coro);

  // Return a future to me from the non-detached edition
  return coro.promise().get_future();

resume_detached:
  // When it returns here we are resumed into the detached edition
}

// 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 this instead:

```
namespace std { namespace coroutines {
  inline auto await#(auto awaitable_expr)
  {
    if(!awaitable_expr.await_ready())
    {
      // affects the calling function, this macro function has no linkage
      coroutine_handle<promise_type> coro = __builtin_coro_suspend(resume_detached);
      __unique.await_suspend(coro);

      // Caller returns future to suspended coroutine
      return -> coro.promise().get_future();
resume_detached:
    }
    // Macro function returns result from await
    return awaitable_expr.await_resume();
  }
}}
```

And voilá, `std::coroutines::await#()` nicely replaces `co_await` in a much more flexible, **entirely library defined**, fashion. Similar macro functions can be written for `std::coroutines::yield#()` and `std::coroutines::return#()`. No core C++ language changes with new keywords needed.

2.2.2 Implementing ranged `for` using these native C++ macro functions

Ranged for loops are already in the language, so this is a bit of a moot point. But let's see how one might implement them with macro functions anyway:

```
namespace std {
  inline void item#(auto &&iterable)
  {
    // Store the current iterator in my macro local scope
    // attached to my calling function
    auto -> it = iterable.begin();

    // Inject a suitable for loop into my calling function
    for -> (; (-> it) != iterable.end(); ++(-> it));

    // As that was a control flow taking a target, we also
    // injected a scope around the statement following the
    // macro invocation.
  }
  inline auto &item#()
  {
    // We get the same macro local scope as any other item#()
    // overload in the same namespace
    return *(-> it);
  }
  inline void foreach#(auto &&iterable)
  {
    // Only item# can see state it itself injected
    item#(iterable);
  }
}
```

And a sample of usage:

```
// Usage
std::vector<std::vector<int>> c;

foreach#(c)



{
  auto &row = item#();

  foreach#(row)

  {
    std::cout << item#() << ", ";
  }

  std::cout << std::endl;
}
//
```

```
// Expanded
std::vector<std::vector<int>> c;
auto __unique1 = c.begin();
for(; __unique1 != c.end(); ++__unique1)
// Extra scope injected by macro injection
// of for loop
{
  {
    auto &row = *__unique1;
    auto __unique2 = row.begin();
    for(; __unique2 != row.end(); ++__unique2)
    {
      {
        std::cout << *__unique2 << ", ";
      }
    }
    std::cout << std::endl;
  }
}
```

This `foreach#()` implementation has some drawbacks however. The most obvious is that if you nest two foreach loops, `item#()` will only return the current item for the innermost loop. This is easily worked around by binding a reference to the current item for outer loops as one goes, but it isn't as clean as with ranged for loops built into the language.

That said, there is clearly lots of power and flexibility available here, and without all the problems that C macros have. The EWG I think will see lots more boilerplate injecting requests landing before it in the next few years. Adding a facility like native C++ macro functions could eliminate much of that workload in my opinion, and enable C++ to evolve onwards without being blocked as much on WG21's capacity limits.

2.2.3 Implementing `try` using these native C++ macro functions

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

```
template <class T, class E>
inline auto try#(std::expected<T, E> v)
{
  // If there is an error, propagate that error immediately
  // by injecting an immediate return of the error into my caller
  if(!v.has_value())
    return -> std::unexpected<E>(std::move(v).error());

  // Otherwise the output of this macro is the value.
  return 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);

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

Similar to the ranged for example implementation, this isn't *quite* as clean looking as the earlier `operator try`, but it sure beats `TRYX(expr)`.

3 Conclusions

3.1 Please propose a better solution!

Above I have proposed two methods of implementing `operator try`. One is narrowly scoped, focused and highly uncontroversial at least in the form of implementation. The other has a very wide scope which could help C++ users all over the world avoid the evils of the C preprocessor for injecting boilerplate, a very common use case.

If you think you have a better idea for dealing with boilerplate injection, please do absolutely consider submitting a P-paper here which shows that both of these proposals are an inferior design. I'll be more than interested to see what you propose.

3.2 Try chaining and Optional

I have not discussed in this proposal paper 'try chaining' at all. This is a facility whereby the try operator can be told to act recursively, repeatedly trying the result of a try until a non-tryable result is encountered.

The reason that I have not discussed it here is because recursive unwrapping is definitely the most useful with Optional, and not in practice that common with Expected|Outcome as in my personal experience, one does not frequently actually store variables in unwrapped form like you do with

Optional. The reason that you do not is because an Expected|Outcome returns an error instead of a value, and more often than not, you want to know about that error sooner rather than later, so you might as well unwrap it. Optional, on the other hand, is a maybe-something, so you tend to use it as a 'slot' and thus write logic based on slots being filled or not. I find myself not using Outcome as 'slots' in over three years of writing code with it. Maybe that's just me of course, but I suspect that try chaining really is mostly useful with Optional not Expected. And if so, perhaps a visitor-based library function would be a better solution to a chaining-capable `operator try`. I'll leave that for readers to decide.

Incidentally, this insight that you won't use Expected like Optional is one of the key reasons why Outcome deviates in design and focus from Expected, and you can find out more about that in *P0762R0 Concerns about Expected* also in this mailing.

3.3 The elephant in the room

Adding any `operator try` at all to the language is of course controversial. You may have noticed my studious avoidance of ever mentioning the phrase 'lightweight error handling' in anything mentioned in this paper. If I had mentioned that phrase, I would be compelled to discuss performance, and to both demonstrate and **prove** the performance superiority of Expected/Outcome based error handling over table-based exception throws where failure is not extremely rare.

However error handling performance cannot be proven with cheap-to-implement micro-benchmarks. It requires you to Monte-Carlo multiple real world code bases and to generate lots of statistical profiles. As it happens, I am also a fully trained Economist, and such an Econometric analysis would not be hard to implement. But it would take lots of time, at least a thousand hours to do it properly by my reckoning. And I'm not willing to invest that sort of time without earning for it, I do earn by the hour as a contract programmer after all.

But I am hoping that this paper inspires someone with much more resources available to them than I have to invest in conclusively proving the merit of this approach. After all, the people over at Swift – many of them former WG21 and Boost members I might add – know their stuff. WG21 choosing to replicate their choice for error handling via Expected and `operator try` is likely not a bad choice.

We just need someone with deep pockets, or an awful lot of free time, to generate the statistical proof of the merits this approach so WG21 can sign off on it. In case you are such a person and are wondering just how much work this actually is, John Lakos has been doing talks during 2017 with performance 'heat maps' of local allocators i.e. colour coded statistical distributions of performance for various canned use cases with various allocators. It took Bloomberg a considerable investment of manpower and time to generate these. A similar level of investment would be needed here too if one is to generate a conclusive proof sufficient to eliminate all doubt in WG21 as to the merit of this approach.

4 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++ macro functions' idea, particularly Ville Voutilainen and Nicol Bolas.
- Michael Park for making available this LaTeX template at https://github.com/mpark/wg21/.

5 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