D0779R0: Proposing `operator try()`

Document #:	D0779R0 draft 1
Date:	2017-09-27
Project:	Programming Language C++ Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com> Vicente J. Botet Escribá <vicente.botet@nokia.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 the Coroutines TS [N4680] provides customisation points for its operator keywords.

Motivating example:

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

expected<int> get_int() noexcept;

expected<std::string> get_int_string() noexcept
{
  try
  {
    expected<int> _int = get_int();
    if(!_int)
      return unexpected(_int.error());
    return std::to_string(*_int);
  }
  catch(...)
  {
    return error_code_from_exception(
      std::current_exception());
  }
}
```

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

expected<int> get_int() noexcept;

expected<std::string> get_int_string() noexcept
{
  try
  {




    return std::to_string(try get_int());
  }
  catch(...)
  {
    return error_code_from_exception(
      std::current_exception());
  }
}
```

This use of the `try` operator approximates, but is not commensurate with, the `try!` macro in Rust (`https://doc.rust-lang.org/std/macro.try.html`) and the `try` keyword in Swift (`https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html`).

1 Potential standard library implementation

```
// Exposition only: Concept for a value|error type.
// Both Expected and Outcome match this concept.
template<typename T>
concept ValueOrError = requires(T a) {
  { a.has_value(); a.value(); a.error(); };
};

// For const lvalue refs
template <ValueOrError T>
constexpr auto operator try(const T &v) noexcept
{
  struct tryer
  {
    const T &v;

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

// For rvalue refs
template <ValueOrError T>
constexpr auto operator try(T &&v) noexcept
{
  struct tryer
  {
    T &&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) };
}


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

expected<int> get_int() noexcept;

expected<std::string> get_int_string() noexcept
{
  try
  {
    return std::to_string(
    try get_int()  /* -->
        auto &&__tryer = operator try (get_int());
        if(__tryer.try_return_immediately())
          return __tryer.try_return_value();
```

```
    <-- __tryer.try_value();
    */
    );
  }
  catch(...)
  {
    return error_code_from_exception(
      std::current_exception());
  }
}
```

2 Proposed standards wording

1. The keyword `try` not followed by a `{` nor a `:` token shall be considered a *try-expression*.
2. Evaluation of a *try-expression* involves the following auxiliary types, expressions, and objects:
   - *a* is the cast-expression.
   - *o* is determined by enumerating the applicable `operator try` functions for an argument *a* (13.3.1.2), and choosing the best one through overload resolution (13.3). If overload resolution is ambiguous, or no viable functions are found, the program is ill-formed. *o* is a call to the selected function.
   - *e* is a temporary object copy-initialized from *o* if *o* is a prvalue; otherwise *e* is an lvalue referring to the result of evaluating *o*.
   - *try-return-immediately* is the expression *e*`.try_return_immediately()`, contextually converted to bool.
   - *try-return-value* is the expression *e*`.try_return_value()`.
   - *try-value* is the expression *e*`.try_value()`.
3. The *try-expression* has the same type and value category as the *try-value* expression.
4. The *try-expression* evaluates the *try-return-immediately* expression, then:
   - If the `result` is true, the current function is exited immediately as if `return` *try-return-value* had been executed.
   - If the `result` is false, the *try-value* expression is evaluated, and its result is the result of the *try-expression*.

3 Acknowledgements

- 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