Document number:	P0323R3
Date:	2017-10-15
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com> JF Bastien <jfbastien@apple.com>

Utility class to represent expected object

Abstract

Class template `expected<T, E>` is a *vocabulary type* which contains an expected value of type `T`, or an error `E`. The class skews towards behaving like a `T`, because its intended use is when the expected type is contained. When something unexpected occurs, more typing is required. When all is good, code mostly looks as if a `T` were being handled.

Table of Contents

Introduction

Class template `expected<T, E>` contains either:

- A value of type `T`, the expected value type; or
- A value of type `E`, an error type used when an unexpected outcome occured.

The interface can be queried as to whether the underlying value is the expected value (of type `T`) or an unexpected value (of type `E`). The original idea comes from Andrei Alexandrescu C++ and Beyond 2012: Systematic Error Handling in C++ Alexandrescu.Expected. The interface and the rational are based on `std::optional` N3793. We consider `expected<T, E>` as a supplement to `optional<T>`, expressing why an expected value isn't contained in the object.

P0262R0 is a related proposal for status / optional value. P0157R0 describes when to use each of the different error report mechanism.

Motivation

C++'s two main error mechanisms are exceptions and return codes. Characteristics of a good error mechanism are:

1. **Error visibility**: Failure cases should appear throughout the code review: debugging can be painful if errors are hidden.
2. **Information on errors**: Errors should carry information from their origin, causes and possibly the ways to resolve it.
3. **Clean code**: Treatment of errors should be in a separate layer of code and as invisible as possible. The reader could notice the presence of exceptional cases without needing to stop reading.
4. **Non-Intrusive error** Errors should not monopolize the communication channel dedicated to normal code flow. They must be as discrete as possible. For instance, the return of a function is a channel that should not be exclusively reserved for errors.

The first and the third characteristic seem contradictory and deserve further explanation. The former points out that errors not handled should appear clearly in the code. The latter tells us that error handling must not interfere with the legibility, meaning that it clearly shows the normal execution flow. Here is a comparison between the exception and return codes:

| | Exception | Return error code | |-|-----------|-------------------| | **Visibility** | Not visible without further analysis of the code. However, if an exception is thrown, we can follow the stack trace. | Visible at the first sight by watching the prototype of the called function. However ignoring return code can lead to undefined results and it can be hard to figure out the problem. | | **Informations** | Exceptions can be arbitrarily rich. | Historically a simple integer. Nowadays, the header `<system_error>` provides richer error code. | | **Clean code** |

Provides clean code, exceptions can be completely invisible for the caller. | Force you to add, at least, a `if` statement after each function call. | | **Non-Intrusive** | Proper communication channel. | Monopolization of the return channel. |

Expected class

We can do the same analysis for the `expected<T, E>` class and observe the advantages over the classic error reporting systems.

1. **Error visibility**: It takes the best of the exception and error code. It's visible because the return type is `expected<T, E>` and users cannot ignore the error case if they want to retrieve the contained value.
2. **Information**: Arbitrarily rich.
3. **Clean code**: The monadic interface of `expected` provides a framework delegating the error handling to another layer of code. Note that `expected<T, E>` can also act as a bridge between an exception-oriented code and a nothrow world.
4. **Non-Intrusive** Use the return channel without monopolizing it.

Other notable characteristics of `expected<T, E>` include:

- Associates errors with computational goals.
- Naturally allows multiple errors inflight.
- Teleportation possible.
- Across thread boundaries.
- On weak executors which don't support thread-local storage.
- Across no-throw subsystem boundaries.
- Across time: save now, throw later.
- Collect, group, combine errors.
- Much simpler for a compiler to optimize.

Use cases

Safe division

This example shows how to define a safe divide operation checking for divide-by-zero conditions. Using exceptions, we might write something like this:

```
struct DivideByZero : public std::exception {...};
double safe_divide(double i, double j)
{
    if (j == 0) throw DivideByZero();
    else return i / j;
}
```

With `expected<T, E>`, we are not required to use exceptions, we can use `std::errc` which is easier to introspect than `std::exception_ptr` if we want to use the error. For the purpose of this example, we use the following enumeration:

```
enum class arithmetic_errc
{
    divide_by_zero, // 9 / 0 == ?
    not_integer_division, // 5 / 2 == 2.5 (which is not an integer)
    integer_divide_overflows, // INT_MIN / -1
};
```

Using `expected<double, errc>`, the code becomes:

```
expected<double, errc> safe_divide(double i, double j)
{
    if (j == 0) return unexpected(arithmetic_errc::divide_by_zero); // (1)
    else return i / j; // (2)
}
```

Note (1) the implicit conversion from `unexpected<E>` to `expected<T, E>` and (2) from `T` to `expected<T, E>` avoid boilerplate code. We have a clean way to fail without using the exception machinery, and we can give precise information about why it failed as well. The liability is that this function is going to be tedious to use. For instance, the exception-based function `i + j / k` is:

```
double f1(double i, double j, double k)
{
    return i + safe_divide(j,k);
}
```

but becomes using `expected<double, errc>` :

```
expected<double, errc> f1(double i, double j, double k)
{
    auto q = safe_divide(j, k);
    if (q) return i + *q;
    else return q;
}
```

We can use `expected<T, E>` to represent different error conditions. For instance, with integer division, we might want to fail if there's overflow, or if the two numbers are not evenly divisible, as well as checking for division by zero. We can overload our `safe_divide` function accordingly:

```
expected<int, errc> safe_divide(int i, int j)
{
    if (j == 0) return unexpected(arithmetic_errc::divide_by_zero);
    if (i == INT_MIN && j == -1) return unexpected(arithmetic_errc::integer_divide_overflows);
    if (i % j != 0) return unexpected(arithmetic_errc::not_integer_division);
    else return i / j;
}
```

URL parsing

The following is how WebKit-based browsers [parse URLs]() and use `std::expected` to denote failure:

```cpp
template<typename CharacterType>
std::expected<uint32_t, URLParser::IPv4PieceParsingError>
URLParser::parseIPv4Piece(CodePointIterator<CharacterType>& iterator, bool& didSeeSyntaxViolation)
{
    enum class State : uint8_t { UnknownBase, Decimal, OctalOrHex, Octal, Hex };
    State state = State::UnknownBase;
    wtf::checked<uint32_t, RecordOverflow> value = 0; // A class for security-critical checked arithmetic.
    if (!iterator.atEnd() && *iterator == '.')
        return std::make_unexpected(IPv4PieceParsingError::Failure);
    while (!iterator.atEnd()) {
        if (isTabOrNewline(*iterator)) {
            didSeeSyntaxViolation = true;
            ++iterator;
            continue;
        }
        if (*iterator == '.') {
            assert(!value.hasOverflowed());
            return value.unsafeGet();
        }
        switch (state) {
        case State::UnknownBase:
            if (*iterator == '0') {
                ++iterator;
                state = State::OctalOrHex;
                break;
            }
            state = State::Decimal;
            break;
        case State::OctalOrHex:
            didSeeSyntaxViolation = true;
            if (*iterator == 'x' || *iterator == 'X') {
                ++iterator;
                state = State::Hex;
                break;
            }
            state = State::Octal;
            break;
        case State::Decimal:
            if (!isASCIIDigit(*iterator))
                return std::make_unexpected(IPv4PieceParsingError::Failure);
            value *= 10;
            value += *iterator - '0';
            if (value.hasOverflowed())
                return std::make_unexpected(IPv4PieceParsingError::Overflow);
            ++iterator;
            break;
        case State::Octal:
            assert(didSeeSyntaxViolation);
            if (*iterator < '0' || *iterator > '7')
                return std::make_unexpected(IPv4PieceParsingError::Failure);
            value *= 8;
            value += *iterator - '0';
            if (value.hasOverflowed())
                return std::make_unexpected(IPv4PieceParsingError::Overflow);
            ++iterator;
            break;
        case State::Hex:
            assert(didSeeSyntaxViolation);
            if (!isASCIIHexDigit(*iterator))
                return std::make_unexpected(IPv4PieceParsingError::Failure);
            value *= 16;
            value += toASCIIHexValue(*iterator);
            if (value.hasOverflowed())
                return std::make_unexpected(IPv4PieceParsingError::Overflow);
            ++iterator;
            break;
        }
    }
    assert(!value.hasOverflowed());
    return value.unsafeGet();
}
```

These results are then accumulated in a `vector`, and different failure conditions are handled differently. An important fact to internalize is that the first failure encountered isn't necessarily the one which is returned, which is why exceptions aren't a good solution here: parsing must continue.

```
template<typename CharacterTypeForSyntaxViolation, typename CharacterType>
std::expected<URLParser::IPv4Address, URLParser::IPv4ParsingError>
URLParser::parseIPv4Host(const CodePointIterator<CharacterTypeForSyntaxViolation>& iteratorForSyntaxViolationPosition, CodePointIterator<Char
{
    std::vector<std::expected<uint32_t, URLParser::IPv4PieceParsingError>> items;
    bool didSeeSyntaxViolation = false;
    if (!iterator.atEnd() && *iterator == '.')
        return std::make_unexpected(IPv4ParsingError::NotIPv4);
    while (!iterator.atEnd()) {
        if (isTabOrNewline(*iterator)) {
            didSeeSyntaxViolation = true;
            ++iterator;
            continue;
        }
        if (items.size() >= 4)
            return std::make_unexpected(IPv4ParsingError::NotIPv4);
        items.append(parseIPv4Piece(iterator, didSeeSyntaxViolation));
        if (!iterator.atEnd() && *iterator == '.') {
            ++iterator;
            if (iterator.atEnd())
                syntaxViolation(iteratorForSyntaxViolationPosition);
            else if (*iterator == '.')
                return std::make_unexpected(IPv4ParsingError::NotIPv4);
        }
    }
    if (!iterator.atEnd() || !items.size() || items.size() > 4)
        return std::make_unexpected(IPv4ParsingError::NotIPv4);
    for (const auto& item : items) {
        if (!item.hasValue() && item.error() == IPv4PieceParsingError::Failure)
            return std::make_unexpected(IPv4ParsingError::NotIPv4);
    }
    for (const auto& item : items) {
        if (!item.hasValue() && item.error() == IPv4PieceParsingError::Overflow)
            return std::make_unexpected(IPv4ParsingError::Failure);
    }
    if (items.size() > 1) {
        for (size_t i = 0; i < items.size() - 1; i++) {
            if (items[i].value() > 255)
                return std::make_unexpected(IPv4ParsingError::Failure);
        }
    }
    if (items[items.size() - 1].value() >= pow256(5 - items.size()))
        return std::make_unexpected(IPv4ParsingError::Failure);

    if (didSeeSyntaxViolation)
        syntaxViolation(iteratorForSyntaxViolationPosition);
    for (const auto& item : items) {
        if (item.value() > 255)
            syntaxViolation(iteratorForSyntaxViolationPosition);
    }

    if (items.size() != 4)
        syntaxViolation(iteratorForSyntaxViolationPosition);

    IPv4Address ipv4 = items.takeLast().value();
    for (size_t counter = 0; counter < items.size(); ++counter)
        ipv4 += items[counter].value() * pow256(3 - counter);
    return ipv4;
}
```

Note that the above code uses `std::make_unexpected` because WebKit still uses C++14. A C++17 codebase would use deduction guides and `std::unexpected` directly.

Error retrieval and correction

The major advantage of `expected<T, E>` over `optional<T>` is the ability to transport an error. Programmer do the following when a function call returns an error:

1. Ignore it.
2. Delegate the responsibility of error handling to higher layer.
3. Try to resolve the error.

Because the first behavior might lead to buggy application, we ignore the usecase. The handling is dependent of the underlying error type, we consider the `exception_ptr` and the `errc` types.

A first imperative way to use our error is to simply extract it from the `expected` using the `error()` member function. The following example shows a `divide2` function that return `0` if the error is `divide_by_zero` :

```
int divide2(int i, int j)
{
    auto e = safe_divide(i, j);
    if (!e)
        switch (e.error().value()) {
        case arithmetic_errc::divide_by_zero: return 0;
        case arithmetic_errc::not_integer_division: return i / j; // Ignore.
        case arithmetic_errc::integer_divide_overflows: return INT_MIN;
        // No default! Adding a new enum value causes a compiler warning here,
        // forcing an update of the code.
        }
    return *e;
}
```

Impact on the standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++17.

Design rationale

The same rationale described in N3672 for `optional<T>` applies to `expected<T, E>` and `expected<T, nullopt_t>` should behave almost the same as `optional<T>` (though we advise using `optional` in that case). The following sections presents the rationale in N3672 applied to `expected<T, E>` .

Conceptual model of `expected<T, E>`

`expected<T, E>` models a discriminated union of types `T` and `unexpected<E>` . `expected<T, E>` is viewed as a value of type `T` or value of type `unexpected<E>` , allocated in the same storage, along with observers to determine which of the two it is.

The interface in this model requires operations such as comparison to `T` , comparison to `E` , assignment and creation from either. It is easy to determine what the value of the expected object is in this model: the type it stores (`T` or `E`) and either the value of `T` or the value of `E` .

Additionally, within the affordable limits, we propose the view that `expected<T, E>` extends the set of the values of `T` by the values of type `E` . This is reflected in initialization, assignment, ordering, and equality comparison with both `T` and `E` . In the case of `optional<T>` , `T` cannot be a `nullopt_t` . As the types `T` and `E` could be the same in `expected<T, E>` , there is need to tag the values of `E` to avoid ambiguous expressions. The `unexpected(E)` deduction guide is proposed for this purpose. However `T` cannot be `unexpected<E>` for a given `E` .

```
expected<int, string> ei = 0;
expected<int, string> ej = 1;
expected<int, string> ek = unexpected(string());

ei = 1;
ej = unexpected(E());;
ek = 0;

ei = unexpected(E());;
ej = 0;
ek = 1;
```

Default `E` template paremeter type

At the Toronto meeting LEWG decided against having a default `E` template parameter type (`std::error_code` or other). This prevents us from providing an `expected` deduction guides for error construction which is a *good thing*: an error was **not** expected, a `T` was expected, it's therefore sensible to force spelling out unexpected outcomes when generating them.

Initialization of `expected<T, E>`

In cases where `T` and `E` have value semantic types capable of storing `n` and `m` distinct values respectively, `expected<T, E>` can be seen as an extended `T` capable of storing `n + m` values: these `T` and `E` stores. Any valid initialization scheme must provide a way to put an expected object to any of these states. In addition, some `T` s aren't `CopyConstructible` and their expected variants still should be constructible with any set of arguments that work for `T` .

As in N3672, the model retained is to initialize either by providing an already constructed `T` or a tagged `E` . The default constructor required `T` to be default-constructible (since `expected<T>` should behave like `T` as much as possible).

```
string s"STR";

expected<string, errc> es{s}; // requires Copyable<T>
expected<string, errc> et = s; // requires Copyable<T>
expected<string, errc> ev = string"STR"; // requires Movable<T>

expected<string, errc> ew; // expected value
expected<string, errc> ex{}; // expected value
expected<string, errc> ey = {}; // expected value
expected<string, errc> ez = expected<string,errc>{}; // expected value
```

In order to create an unexpected object, the deduction guide `unexpected` needs to be used:

```
expected<string, int> ep{unexpected(-1)}; // unexpected value, requires Movable<E>
expected<string, int> eq = unexpected(-1); // unexpected value, requires Movable<E>
```

As in N3672, and in order to avoid calling move/copy constructor of `T`, we use a “tagged” placement constructor:

```
expected<MoveOnly, errc> eg; // expected value
expected<MoveOnly, errc> eh{}; // expected value
expected<MoveOnly, errc> ei{in_place}; // calls MoveOnly{} in place
expected<MoveOnly, errc> ej{in_place, "arg"}; // calls MoveOnly{"arg"} in place
```

To avoid calling move/copy constructor of `E`, we use a “tagged” placement constructor:

```
expected<int, string> ei{unexpect}; // unexpected value, calls string{} in place
expected<int, string> ej{unexpect, "arg"}; // unexpected value, calls string{"arg"} in place
```

An alternative name for `in_place` that is coherent with `unexpect` could be `expect`. Being compatible with `optional<T>` seems more important. So this proposal doesn't propose such an `expect` tag.

The alternative and also comprehensive initialization approach, which is compatible with the default construction of `expected<T, E>` as `T()`, could have been a variadic perfect forwarding constructor that just forwards any set of arguments to the constructor of the contained object of type `T`.

Never-empty guarantee

As for `boost::variant<T, unexpected<E>>`, `expected<T, E>` ensures that it is never empty. All instances `v` of type `expected<T, E>` guarantee `v` has constructed content of one of the types `T` or `E`, even if an operation on `v` has previously failed.

This implies that `expected` may be viewed precisely as a union of exactly its bounded types. This "never-empty" property insulates the user from the possibility of undefined `expected` content or an `expected` `valueless_by_exception` as `std::variant` and the significant additional complexity-of-use attendant with such a possibility.

In order to ensure this property the types `T` and `E` must satisfy the requirements as described in P0110R0. Given the nature of the parameter `E`, that is, to transport an error, it is expected to be `is_nothrow_copy_constructible<E>`, `is_nothrow_move_constructible<E>`, `is_nothrow_copy_assignable<E>` and `is_nothrow_move_assignable<E>`.

Note however that these constraints are applied only to the operations that need them.

If `is_nothrow_constructible<T, Args...>` is `false`, the `expected<T, E>::emplace(Args...)` function is not defined. In this case, it is the responsibility of the user to create a temporary and move or copy it.

The default constructor

Similar data structure includes `optional<T>`, `variant<T1,...,Tn>` and `future<T>`. We can compare how they are default constructed.

- `std::optional<T>` default constructs to an optional with no value.
- `std::variant<T1,...,Tn>` default constructs to `T1` if default constructible or it is ill-formed
- `std::future<T>` default constructs to an invalid future with no shared state associated, that is, no value and no exception.
- `std::optional<T>` default constructor is equivalent to `boost::variant<nullopt_t, T>`.

This raises several questions about `expected<T, E>`:

- Should the default constructor of `expected<T, E>` behave like `variant<T, unexpected<E>>` or as `variant<unexpected<E>,T>`?
- Should the default constructor of `expected<T, nullopt_t>` behave like `optional<T>`? If yes, how should behave the default constructor of `expected<T, E>`? As if initialized with `unexpected(E())`? This would be equivalent to the initialization of `variant<unexpected<E>,T>`.
- Should `expected<T, E>` provide a default constructor at all? N3527 presents valid arguments against this approach, e.g. `array<expected<T, E>>` would not be possible.

Requiring `E` to be default constructible seems less constraining than requiring `T` to be default constructible (e.g. consider the `Date` example in N3527). With the same semantics `expected<Date,E>` would be `Regular` with a meaningful not-a-date state created by default.

The authors consider the arguments in N3527 valid for `optional<T>` and `expected<T, E>`, however the committee requested that `expected<T, E>` default constructor should behave as constructed with `T()` if `T` is default constructible.

Could `Error` be `void`

`void` isn't a sensible `E` template parameter type: the `expected<T, void>` vocabulary type means "I expect a `T`, but I may have nothing for you". This is literally what `optional<T>` is for. If the error is a unit type the user can use `std::monostate` or `std::nullopt`.

Conversion from `T`

An object of type `T` is implicitly convertible to an expected object of type `expected<T, E>`:

```
expected<int, errc> ei = 1; // works
```

This convenience feature is not strictly necessary because you can achieve the same effect by using tagged forwarding constructor:

```
expected<int, errc> ei{in_place, 1};
```

It has been demonstrated that this implicit conversion is dangerous a-gotcha-with-optional.

An alternative will be to make it explicit and add a `success<T>` (similar to `unexpected<E>` explicitly convertible from `T` and implicitly convertible to `expected<T, E>`.

```
expected<int, errc> ei = success(1);
expected<int, errc> ej = unexpected(ec);
```

The authors consider that it is safer to have the explicit conversion, the implicit conversion is so friendly that we don't propose yet an explicit conversion. In addition `std::optional` has already be delivered in C++17 and it has this gotcha.

Further, having `success` makes code much more verbose than the current implicit conversion. Forcing the usage of `success` would make `expected` a much less useful vocabulary type: if success is expected then success need not be called out.

Conversion from `E`

An object of type `E` is not convertible to an unexpected object of type `expected<T, E>` since `E` and `T` can be of the same type. The proposed interface uses a special tag `unexpect` and `unexpected` deduction guide to indicate an unexpected state for `expected<T, E>`. It is used for construction and assignment. This might raise a couple of objections. First, this duplication is not strictly necessary because you can achieve the same effect by using the `unexpect` tag forwarding constructor:

```
expected<string, errc> exp1 = unexpected(1);
expected<string, errc> exp2 = {unexpect, 1};
exp1 = unexpected(1);
exp2 = {unexpect, 1};
```

or simply using deduced template parameter for constructors

```
expected<string, errc> exp1 = unexpected(1);
exp1 = unexpected(1);
```

While some situations would work with the `{unexpect, ...}` syntax, using `unexpected` makes the programmer's intention as clear and less cryptic. Compare these:

```
expected<vector<int>, errc> get1() {}
    return {unexpect, 1};
}
expected<vector<int>, errc> get2() {
    return unexpected(1);
}
expected<vector<int>, errc> get3() {
    return expected<vector<int>, int>{unexpect, 1};
}
expected<vector<int>, errc> get2() {
    return unexpected(1);
}
```

The usage of `unexpected` is also a consequence of the adapted model for `expected`: a discriminated union of `T` and `unexpected<E>`.

Should we support the `exp2 = {}` ?

Note also that the definition of `unexpected` has an explicitly deleted default constructor. This was in order to enable the reset idiom `exp2 = {}` which would otherwise not work due to the ambiguity when deducing the right-hand side argument.

Now that `expected<T, E>` defaults to `T{}` the meaning of `exp2 = {}` is to assign `T{}` .

Observers

In order to be as efficient as possible, this proposal includes observers with narrow and wide contracts. Thus, the `value()` function has a wide contract. If the expected object does not contain a value, an exception is thrown. However, when the user knows that the expected object is valid, the use of `operator*` would be more appropriated.

Explicit conversion to `bool`

The rational described in N3672 for `optional<T>` applies to `expected<T, E>` . The following example therefore combines initialization and value-checking in a boolean context.

```
if (expected<char, errc> ch = readNextChar()) {
// ...
}
```

`has_value()` following P0032

`has_value()` has been added to follow [P0032R2].

Accessing the contained value

Even if `expected<T, E>` has not been used in practice for enough time as `std::optional` or Boost.Optional, we consider that following the same interface as `std::optional<T>` makes the C++ standard library more homogeneous.

The rational described in N3672 for `optional<T>` applies to `expected<T, E>` .

Dereference operator

The indirection operator was chosen because, along with explicit conversion to `bool` , it is a very common pattern for accessing a value that might not be there:

```
if (p) use(*p);
```

This pattern is used for all sort of pointers (smart or raw) and `optional` ; it clearly indicates the fact that the value may be missing and that we return a reference rather than a value. The indirection operator created some objections because it may incorrectly imply `expected` and `optional` are a (possibly smart) pointer, and thus provides shallow copy and comparison semantics. All library components so far use indirection operator to return an object that is not part of the pointer's/iterator's value. In contrast, `expected` as well as `optional` indirects to the part of its own state. We do not consider it a problem in the design; it is more like an unprecedented usage of indirection operator. We believe that the cost of potential confusion is overweighted by the benefit of an intuitive interface for accessing the contained value.

We do not think that providing an implicit conversion to `T` would be a good choice. First, it would require different way of checking for the empty state; and second, such implicit conversion is not perfect and still requires other means of accessing the contained value if we want to call a member function on it.

Using the indirection operator for an object that does not contain a value is undefined behavior. This behavior offers maximum runtime performance.

Function value

In addition to the indirection operator, we propose the member function `value` as in N3672 that returns a reference to the contained value if one exists or throw an exception otherwise.

```
void interact() {
    string s;
    cout << "enter number: ";
    cin >> s;
    expected<int, error> ei = str2int(s);
    try {
        process_int(ei.value());
    }
    catch(bad_expected_access<error>) {
        cout << "this was not a number.";
    }
}
```

The exception thrown is `bad_expected_access<E>` (derived from `std::exception`) which will contain the stored error.

`bad_expected_access<E>` and `bad_optional_access` could inherit both from a `bad_access` exception derived from `exception`, but this is not proposed yet.

Should `expected<T, E>::value()` throw `E` instead of `bad_expected_access<E>`?

As any type can be thrown as an exception, should `expected<T, E>` throw `E` instead of `bad_expected_access<E>`?

Some argument that standard function should throw exceptions that inherit from `std::exception`, but here the exception throw is given by the user via the type `E`, it is not the standard library that throws explicitly an exception that don't inherit from `std::exception`.

This could be convenient as the user will have directly the `E` exception. However it will be more difficult to identify that this was due to a bad expected access.

If yes, should `optional<T>` throw `nullopt_t` instead of `bad_optional_access` to be coherent?

We don't propose this.

Other have suggested to throw `system_error` if `E` is `error_code`, rethrow if `E` is `exception_ptr`, `E` if it inherits from `std::exception` and `bad_expected_access<E>` otherwise.

An alternative would be to add some customization point that state which exception is thrown but we don't propose it in this proposal. See the Appendix I.

Accessing the contained error

Usually, accessing the contained error is done once we know the expected object has no value. This is why the `error()` function has a narrow contract: it works only if `*this` does not contain a value.

```
expected<int, errc> getIntOrZero(istream_range& r) {
    auto r = getInt(); // won't throw
    if (!r && r.error() == errc::empty_stream) {
        return 0;
    }
    return r;
}
```

This behavior could not be obtained with the `value_or()` method since we want to return `0` only if the error is equal to `empty_stream`.

We could as well provide an error access function with a wide contract. We just need to see how to name each one.

Conversion to the unexpected value

The `error()` function is used to propagate errors, as for example in the next example:

```
expected<pair<int, int>, errc> getIntRange(istream_range& r) {
    auto f = getInt(r);
    if (!f) return unexpected(f.error());
    auto m = matchedString("..", r);
    if (!m) return unexpected(m.error());
    auto l = getInt(r);
    if (!l) return unexpected(l.error());
    return std::make_pair(*f, *l);
}
```

Function `value_or`

The function member `value_or()` has the same semantics than `optional` [N3672](N3672) since the type of `E` doesn't matter; hence we can consider that `E == nullopt_t` and the `optional` semantics yields.

This function is a convenience function that should be a non-member function for `optional` and `expected`, however as it is already part of the `optional` interface we propose to have it also for `expected`.

Equality operators

As for `optional` and `variant`, one of the design goals of `expected` is that objects of type `expected<T, E>` should be valid elements in STL containers and usable with STL algorithms (at least if objects of type `T` and `E` are). Equality comparison is essential for `expected<T, E>` to model concept `Regular`. C++ does not have concepts yet, but being regular is still essential for the type to be effectively used with STL.

Comparison operators

Comparison operators between `expected` objects, and between mixed `expected` and `unexpected`, aren't required at this time. A future proposal could re-adopt the comparisons as defined in [P0323R2](P0323R2).

Modifiers

Resetting the value

Reseting the value of `expected<T, E>` is similar to `optional<T>` but instead of building a disengaged `optional<T>`, we build an erroneous `expected<T, E>`. Hence, the semantics and rationale is the same than in N3672.

Tag `in_place`

This proposal makes use of the "in-place" tag as defined in [C++17]. This proposal provides the same kind of "in-place" constructor that forwards (perfectly) the arguments provided to `expected`'s constructor into the constructor of `T`.

In order to trigger this constructor one has to use the tag `in_place`. We need the extra tag to disambiguate certain situations, like calling `expected`'s default constructor and requesting `T`'s default construction:

```
expected<Big, error> eb{in_place, "1"}; // calls Big{"1"} in place (no moving)
expected<Big, error> ec{in_place}; // calls Big{} in place (no moving)
expected<Big, error> ed{}; // calls Big{} (expected state)
```

Tag `unexpect`

This proposal provides an "unexpect" constructor that forwards (perfectly) the arguments provided to `expected`'s constructor into the constructor of `E`. In order to trigger this constructor one has to use the tag `unexpect`.

We need the extra tag to disambiguate certain situations, notably if `T` and `E` are the same type.

```
expected<Big, error> eb{unexpect, "1"}; // calls error{"1"} in place (no moving)
expected<Big, error> ec{unexpect}; // calls error{} in place (no moving)
```

In order to make the tag uniform an additional "expect" constructor could be provided but this proposal doesn't propose it.

Requirements on `T` and `E`

Class template `expected` imposes little requirements on `T` and `E`: they have to be complete object type satisfying the requirements of `Destructible`. Each operations on `expected<T, E>` have different requirements and may be disable if `T` or `E` doesn't respect these requirements. For example, `expected<T, E>`'s move constructor requires that `T` and `E` are `MoveConstructible`, `expected<T, E>`'s copy constructor requires that `T` and `E` are `CopyConstructible`, and so on. This is because `expected<T, E>` is a wrapper for `T` or `E`: it should resemble `T` or `E` as much as possible. If `T` and `E` are `EqualityComparable` then (and only then) we expect `expected<T, E>` to be `EqualityComparable`.

However in order to ensure the never empty guaranties, `expected<T, E>` requires `E` to be no throw move constructible. This is normal as the `E` stands for an error, and throwing while reporting an error is a very bad thing.

Expected references

This proposal doesn't include `expected` references as `optional` [C++17] doesn't include references either.

We need a future proposal.

Expected `void`

While it could seem weird to instantiate `optional` with `void`, it has more sense for `expected` as it conveys in addition, as `future<T>`, an error state. The type `expected<void, E>` means "nothing is expected, but an error could occur".

Making expected a literal type

In N3672, they propose to make `optional` a literal type, the same reasoning can be applied to expected. Under some conditions, such that `T` and `E` are trivially destructible, and the same described for `optional`, we propose that `expected` be a literal type.

Moved from state

We follow the approach taken in `optional` N3672. Moving `expected<T, E>` does not modify the state of the source (valued or erroneous) of `expected` and the move semantics is up to `T` or `E`.

I/O operations

For the same reasons as `optional` N3672 we do not add `operator<<` and `operator>>` I/O operations.

What happens when `E` is a status?

When `E` is a status, as most of the error codes are, and has more than one value that mean success, setting an `expected<T, E>` with a successful `e` value could be misleading if the user expect in this case to have also a `T`. In this case the user should use the proposed `status_value<E, T>` class. However, if there is only one value `e` that mean success, there is no such need and `expected<T, E>` compose better with the monadic interface P0650R0.

Do we need an `expected<T, E>::error_or` function?

See P0786R0.

Do we need to add such an `error_or` function? as member?

This function should work for all the *ValueOrError* types and so could belong to a future *ValueOrError* proposal.

Not in this proposal.

Do we need a `expected<T, E>::check_error` function?

See P0786R0.

Do we want to add such a `check_error` function? as member?

This function should work for all the *ValueOrError* types and so could belong to a future *ValueOrError* proposal.

Not in this proposal.

Do we need a `expected<T,G>::adapt_error(function<E(G))` function?

We have the constructor `expected<T, E>(expected<T,G>)` that allows to transport explicitly the contained error as soon as it is convertible.

However sometimes we cannot change either of the error types and we could need to do this transformation. This function help to achieve this goal. The parameter is the function doing the error transformation.

This function can be defined on top of the existing interface.

```
template <class T, class E>
expected<T,G> adapt_error(expected<T, E> const& e, function<G(E)> adaptor) {
    if ( !e ) return adaptor(e.error());
    else return expected<T,G>(*e);
}
```

Do we want to add such a `adapt_error` function? as member?

This function should work for all the *ValueOrError* types and so could belong to a future *ValueOrError* proposal.

Not in this proposal.

Open points

The authors would like to have an answer to the following points:

Do we want it for the IS or for the TS?

The proposed wording is for the Library Fundamental V3 TS. However, there is no dependency on it and the wording can be adapted without major changes to the IS.

What about inherit `bad_expect_access<E>` from `bad_expect_access<void>` ?

This has the advantage to make it easier for the user to manage with any bad access to expected when the user doesn't care of the error.

The same argument can be seen as a bad thing. It is too easy to ignore the error.

In addition `bad_expect_access<E>` should default the `E` parameter to `void`.

This point wasn't discussed enough in Toronto. The authors consider that this should be a correct design.

Do we need separated wording for `expected<void,E>` ?

The authors have started to introduce the wording for `expected<void,E>` specialization as part of the `expected<T,E>` introducing whenever needed special constraints or behavior.

An alternative would consists in duplicating the wording for `expected<void,E>`.

Which style is considered better for the standard wording?

Proposed Wording

The proposed changes are expressed as edits to N4617 the Working Draft - C++ Extensions for Library Fundamentals V2. The wording has been adapted from the section "Optional objects".

General utilities library

--- Insert a new section. ---

X.Y Unexpected objects [[unexpected]]

X.Y.1 In general [unexpected.general]

This subclause describes class template `unexpected` that contain objects representing an unexpected outcome. This unexpected object implicitly convertible to other objects.

X.Y.2 Header synopsis [unexpected.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
    // X.Y.3, Unexpected object type
    template <class E>
      class unexpected;

    // X.Y.5, unexpected relational operators
    template <class E>
        constexpr bool
        operator==(const unexpected<E>&, const unexpected<E>&);
    template <class E>
        constexpr bool
        operator!=(const unexpected<E>&, const unexpected<E>&);

}}}
```

A program that needs the instantiation of template `unexpected` for a reference type or `void` is ill-formed.

X.Y.3 Unexpected object type [unexpected.object]

```
template <class E>
class unexpected {
public:
    unexpected() = delete;
    constexpr explicit unexpected(const E&);
    constexpr explicit unexpected(E&&);
    constexpr const E& value() const &;
    constexpr E& value() &;
    constexpr E&& value() &&;
    constexpr E const&& value() const&&;
private:
    E val; // exposition only
};
```

If `E` is void the program is ill formed.

```
constexpr explicit unexpected(const E&);
```

Effects: Build an `unexpected` by copying the parameter to the internal storage `val`.

```
constexpr explicit unexpected(E &&);
```

Effects: Build an `unexpected` by moving the parameter to the internal storage `val`.

```
constexpr const E& value() const &;
constexpr E& value() &;
```

Returns: `val`.

```
constexpr E&& value() &&;
constexpr E const&& value() const&&;
```

Returns: `move(val)`.

X.Y.5 Unexpected Relational operators [unexpected.relational_op]

```
template <class E>
    constexpr bool operator==(const unexpected<E>& x, const unexpected<E>& y);
```

Requires: `E` shall meet the requirements of *EqualityComparable*.

Returns: `x.value() == y.value()`.

Remarks: Specializations of this function template, for which `x.value() == y.value()` is a core constant expression, shall be constexpr functions.

```
template <class E>
    constexpr bool operator!=(const unexpected<E>& x, const unexpected<E>& y);
```

Requires: `E` shall meet the requirements of *EqualityComparable*.

Returns: `x.value() != y.value()`.

Remarks: Specializations of this function template, for which `x.value() != y.value()` is a core constant expression, shall be constexpr functions.

--- Insert a new section. ---

X.Z Expected objects [[expected]]

X.Z.1 In general [expected.general]

This sub-clause describes class template expected that represents expected objects. An `expected<T, E>` object is an object that contains the storage for another object and manages the lifetime of this contained object `T`, alternatively it could contain the storage for another unexpected object `E`. The contained object may not be initialized after the expected object has been initialized, and may not be destroyed before the expected object has been destroyed. The initialization state of the contained object is tracked by the expected object.

X.Z.2 Header `<experimental/expected>` synopsis [expected.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
    // X.Z.4, expected for object types
    template <class T, class E>
        class expected;

    // X.Z.5, expected specialization for void
    template <class E>
        class expected<void,E>;

    // X.Z.6, unexpect tag
    struct unexpect_t {
       unexpect_t() = default;
    };
    inline constexpr unexpect_t unexpect{};

    // X.Z.7, class bad_expected_access
    template <class E>
       class bad_expected_access;

    // X.Z.8, Specialization for void.
    template <>
       class bad_expected_access<void>;

    // X.Z.9, Expected relational operators
    template <class T, class E>
        constexpr bool operator==(const expected<T, E>&, const expected<T, E>&);
    template <class T, class E>
        constexpr bool operator!=(const expected<T, E>&, const expected<T, E>&);

    // X.Z.10, Comparison with T
    template <class T, class E>
      constexpr bool operator==(const expected<T, E>&, const T&);
    template <class T, class E>
      constexpr bool operator==(const T&, const expected<T, E>&);
    template <class T, class E>
      constexpr bool operator!=(const expected<T, E>&, const T&);
    template <class T, class E>
      constexpr bool operator!=(const T&, const expected<T, E>&);

    // X.Z.10, Comparison with unexpected<E>
    template <class T, class E>
      constexpr bool operator==(const expected<T, E>&, const unexpected<E>&);
    template <class T, class E>
      constexpr bool operator==(const unexpected<E>&, const expected<T, E>&);
    template <class T, class E>
      constexpr bool operator!=(const expected<T, E>&, const unexpected<E>&);
    template <class T, class E>
      constexpr bool operator!=(const unexpected<E>&, const expected<T, E>&);

    // X.Z.11, Specialized algorithms
    void swap(expected<T, E>&, expected<T, E>&) noexcept(see below);

}}}
```

A program that necessitates the instantiation of template `expected<T, E>` with `T` for a reference type or for possibly cv-qualified types `in_place_t`, `unexpect_t` or `unexpected<E>` or `E` for a reference type or `void` is ill-formed.

X.Z.3 Definitions [expected.defs]

An instance of `expected<T, E>` is said to be valued if it contains a value of type `T`. An instance of `expected<T, E>` is said to be unexpected if it contains an object of type `E`.

X.Y.4 expected for object types [expected.object]

```
template <class T, class E>
class expected
{
public:
    typedef T value_type;
    typedef E error_type;
    typedef unexpected<E> unexpected_type;

    template <class U>
```

```
        struct rebind {
        using type = expected<U, error_type>;
      };

    // X.Z.4.1, constructors
    constexpr expected();
    constexpr expected(const expected&);
    constexpr expected(expected&&) noexcept(see below);
    template <class U, class G>
        EXPLICIT constexpr expected(const expected<U, G>&);
    template <class U, class G>
        EXPLICIT constexpr expected(expected<U, G>&&);

    template <class U = T>
        EXPLICIT constexpr expected(U&& v);

    template <class... Args>
        constexpr explicit expected(in_place_t, Args&&...);
    template <class U, class... Args>
        constexpr explicit expected(in_place_t, initializer_list<U>, Args&&...);
    template <class G = E>
        constexpr expected(unexpected<G> const&);
    template <class G = E>
        constexpr expected(unexpected<G> &&);
    template <class... Args>
        constexpr explicit expected(unexpect_t, Args&&...);
    template <class U, class... Args>
        constexpr explicit expected(unexpect_t, initializer_list<U>, Args&&...);

    // X.Z.4.2, destructor
    ~expected();

    // X.Z.4.3, assignment
    expected& operator=(const expected&);
    expected& operator=(expected&&) noexcept(see below);
    template <class U = T> expected& operator=(U&&);
    template <class G = E>
        expected& operator=(const unexpected<G>&);
    template <class G = E>
        expected& operator=(unexpected<G>&&) noexcept(see below);

    template <class... Args>
        void emplace(Args&&...);
    template <class U, class... Args>
        void emplace(initializer_list<U>, Args&&...);

    // X.Z.4.4, swap
    void swap(expected&) noexcept(see below);

    // X.Z.4.5, observers
    constexpr const T* operator ->() const;
    constexpr T* operator ->();
    constexpr const T& operator *() const&;
    constexpr T& operator *() &;
    constexpr const T&& operator *() const &&;
    constexpr T&& operator *() &&;
    constexpr explicit operator bool() const noexcept;
    constexpr bool has_value() const noexcept;
    constexpr const T& value() const&;
    constexpr T& value() &;
    constexpr const T&& value() const &&;
    constexpr T&& value() &&;
    constexpr const E& error() const&;
    constexpr E& error() &;
    constexpr const E&& error() const &&;
    constexpr E&& error() &&;
    template <class U>
        constexpr T value_or(U&&) const&;
    template <class U>
        T value_or(U&&) &&;

private:
    bool has_val; // exposition only
    union
    {
        value_type val; // exposition only
        unexpected_type unexpect; // exposition only
```

```
    };
};
```

Valued instances of `expected<T, E>` where `T` and `E` are of object type shall contain a value of type `T` or a value of type `E` within its own storage. These values are referred to as the contained or the unexpected value of the `expected` object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained or unexpected value. The contained or unexpected value shall be allocated in a region of the `expected<T, E>` storage suitably aligned for the type `T` and `unexpected<E>`. Members `has_val`, `val` and `unexpect` are provided for exposition only. Implementations need not provide those members. `has_val` indicates whether the expected object's contained value has been initialized (and not yet destroyed); when `has_val` is true `val` points to the contained value, and when it is false `unexpect` points to the erroneous value.

`T` must be `void` or shall be object type and shall satisfy the requirements of `Destructible` (Table 27).

`E` shall be object type and shall satisfy the requirements of `Destructible` (Table 27).

X.Z.4.1 Constructors [expected.object.ctor]

```
constexpr expected();
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `T{}` (if `T` is not `void`).

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the default constructor of `T` (nothing if `T` is `void`).

Remarks: If value-initialization of `T` is a constexpr constructor or `T` is `void` this constructor shall be constexpr. This constructor shall be defined as deleted unless `is_default_constructible_v<T>` or `T` is `void`.

```
constexpr expected(const expected& rhs);
```

Effects: If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs` (if `T` is not `void`).

If `rhs` does not contain a value initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `unexpected(rhs.error())`.

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void` or by the selected constructor of `unexpected<E>`.

Remarks: This constructor shall be defined as deleted unless `is_copy_constructible_v<T>` or `T` is `void` and `is_copy_constructible_v<E>`. If `is_trivially_copy_constructible_v<T>` is `true` or `T` is `void` and `is_trivially_copy_constructible_v<E>` is `true`, this constructor shall be a constexpr constructor.

```
constexpr expected(expected && rhs) noexcept('see below');
```

Effects: If `rhs` contains a value initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)` (if `T` is not `void`).

If `rhs` does not contain a value initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `std::move(unexpected(rhs.error()))`.

`bool(rhs)` is unchanged.

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void` or by the selected constructor of `unexpected<E>`.

Remarks: The expression inside `noexcept` is equivalent to: `is_nothrow_move_constructible_v<T> or` T `is` `void` and `is_nothrow_move_constructible_v<E>`. This constructor shall not participate in overload resolution unless `is_move_constructible_v<T> and is_move_constructible_v<E>`. If `is_trivially_move_constructible_v<T>` is `true` or `T` is `void` and `is_trivially_move_constructible_v<E>` is `true`, this constructor shall be a constexpr constructor.

```
    template <class U, class G>
    EXPLICIT constexpr expected(const expected<U,G>& rhs);
```

Effects: If `rhs` contains a value initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs` (if `T` is not `void`).

If `rhs` does not contain a value initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `unexpected(rhs.error())`.

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void` or by the selected constructor of `unexpected<E>`.

Remarks: This constructor shall not participate in overload resolution unless:

- `is_constructible_v<T, const U&>` is `true` or `T` and `U` are `void`,
- `is_constructible_v<E, const G&>` is `true`,
- `is_constructible_v<T, expected<U,G>&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, expected<U,G>&&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, const expected<U,G>&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, const expected<U,G>&&>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<expected<U,G>&, T>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<expected<U,G>&&, T>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<const expected<U,G>&, T>` is `false` or `T` and `U` are `void`, and
- `is_convertible_v<const expected<U,G>&&, T>` is `false` or `T` and `U` are `void`.

The constructor is explicit if and only if `T` is not `void` and `is_convertible_v<U const&, T>` is false or `is_convertible_v<G const&, E>` is false.

```
template <class U, class G>
EXPLICIT constexpr expected(expected<U,G>&& rhs);
```

Effects: If `rhs` contains a value initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)` or nothing if `T` is `void`.

If `rhs` does not contain a value initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `std::move(unexpected(rhs.error()))`. `bool(rhs)` is unchanged

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void` or by the selected constructor of `unexpected<E>`.

Remarks: This constructor shall not participate in overload resolution unless:

- `is_constructible_v<T, U&&>` is `true`,
- `is_constructible_v<E, G&&>` is `true`,
- `is_constructible_v<T, expected<U,G>&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, expected<U,G>&&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, const expected<U,G>&>` is `false` or `T` and `U` are `void`,
- `is_constructible_v<T, const expected<U,G>&&>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<expected<U,G>&, T>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<expected<U,G>&&, T>` is `false` or `T` and `U` are `void`,
- `is_convertible_v<const expected<U,G>&, T>` is `false` or `T` and `U` are `void`, and
- `is_convertible_v<const expected<U,G>&&, T>` is `false` or `T` and `U` are `void`.

The constructor is explicit if and only if `is_convertible_v<U&&, T>` is false or `is_convertible_v<G&&, E>` is false.

```
template <class U = T>
EXPLICIT constexpr expected(U&& v);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::forward<U>(v)`.

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If `T`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `T` is not `void` and `is_constructible_v<T, U&&>` is `true`, `is_same_v<decay_t<U>, in_place_t>` is `false`, `is_same_v<expected<T, E>, decay_t<U>>` is false, and `is_same_v<unexpected<E>, decay_t<U>>` is false. The constructor is explicit if and only if `is_convertible_v<U&&, T>` is false.

```
template <class... Args>
  constexpr explicit expected(in_place_t, Args&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `std::forward<Args>(args)...`.

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void`.

Remarks: If `T`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `T` is not `void` and `is_constructible_v<T, Args&&...>`.

```
template <class U, class... Args>
  constexpr explicit expected(in_place_t, initializer_list<U> il, Args&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected constructor of `T` if `T` is not `void`.

Remarks: If `T`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `T` is not `void` and `is_constructible<T, initializer_list_v<U>&, Args&&...>`.

```
template <class G = E>
EXPLICIT constexpr expected(unexpected<G> const& e);
```

Effects: Initializes the unexpected value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `e`.

Postconditions: `*this` does not contain a value.

Throws: Any exception thrown by the selected constructor of `unexpected<E>`

Remark: If `unexpected<E>`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `is_constructible_v<E, const G&>`. The constructor is explicit if and only if `is_convertible_v<const G&, E>` is false.

```
template <class G = E>
EXPLICIT constexpr expected(unexpected<G>&& e);
```

Effects: Initializes the unexpected value as if direct-non-list-initializing an object of type `unexpected<E>` with the expression `std::move(e)`.

Postconditions: `*this` does not contain a value.

Throws: Any exception thrown by the selected constructor of `unexpected<E>`

Remark: If `unexpected<E>`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor. The expression inside `noexcept` is equivalent to: `is_nothrow_constructible_v<E, G&&>`. This constructor shall not participate in overload resolution unless `is_constructible_v<E, G&&>`. The constructor is explicit if and only if `is_convertible_v<G&&, E>` is false.

```
template <class... Args>
  constexpr explicit expected(unexpect_t, Args&&... args);
```

Effects: Initializes the unexpected value as if direct-non-list-initializing an object of type `unexpected<E>` with the arguments `std::forward<Args>(args)...`.

Postconditions: `*this` does not contain a value.

Throws: Any exception thrown by the selected constructor of `unexpected<E>`

Remarks: If `unexpected<E>`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `is_constructible_v<E, Args&&...>`.

```
template <class U, class... Args>
  constexpr explicit expected(unexpect_t, initializer_list<U> il, Args&&... args);
```

Effects: Initializes the unexpected value as if direct-non-list-initializing an object of type `unexpected<E>` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `*this` does not contain a value.

Throws: Any exception thrown by the selected constructor of `unexpected<E>`.

Remarks: If `unexpected<E>`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless `is_constructible_v<E, initializer_list<U>&, Args&&...>`.

X.Z.4.2 Destructor [expected.object.dtor]

```
~expected();
```

Effects: If `T` is not `void` and `is_trivially_destructible_v<T> != true` and `*this` contains a value, calls `val.~T()`. If `is_trivially_destructible_v<E> != true` and `*this` does not contain a value, calls `unexpect.~unexpected<E>()`.

Remarks: If `T` is `void` or `is_trivially_destructible_v<T> and is_trivially_destructible_v<E>` is `true` then this destructor shall be a trivial destructor.

X.Z.4.3 Assignment [expected.object.assign]

```
expected<T, E>& operator=(const expected<T, E>& rhs) noexcept(see below);
```

Effects:

If `*this` contains a value and `rhs` contains a value, assigns `*rhs` to the contained value `val` if `T` is not `void` ;

otherwise, if `*this` does not contain a value and `rhs` does not contain a value, assigns `unexpected(rhs.error())` to the contained value `unexpect` ;

otherwise, if `*this` contains a value and `rhs` does not contain a value,

- destroys the contained value by calling `val.~T()` if `T` is not `void` ,
- initializes the unexpect value as if direct-non-list-initializing an object of type `unexpected<E>` with `unexpected(rhs.error())` ;

otherwise `*this` does not contain a value and `rhs` contains a value

if `is_nothrow_copy_constructible_v<T>` or `T` is `void`

- destroys the unexpect value by calling `unexpect.~unexpected<E>()`
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs` if `T` is not `void` ;

otherwise, if `is_nothrow_move_constructible_v<T>` (`T` is not `void`)

- constructs a new `T tmp` on the stack from `*rhs` ,
- destroys the contained value by calling `unexpect.~unexpected<E>()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `tmp` ;

otherwise as `is_nothrow_move_constructible_v<E>` (`T` is not `void`)

- move constructs a new `unexpected<E> tmp` on the stack from `this.get_unexpecte()` (which can't throw as `is_nothrow_move_constructible_v<E>` is `true`),
- destroys the contained value by calling `unexpect.~unexpected<E>()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs` . Either,
  - the constructor didn't throw, so mark the expected as holding a `T` (which can't throw), or
  - the constructor did throw, so move-construct the `unexpected<E>` from the stack `tmp` back into the expected storage (which can't throw as `is_nothrow_move_constructible_v<E>` is `true`), and rethrow the exception.

Returns: `*this` .

Postconditions: `bool(rhs) == bool(*this)` .

Remarks: If any exception is thrown, the values of `bool(*this) and bool(rhs)` remain unchanged.

If an exception is thrown during the call to `T` 's copy constructor, no effect. If an exception is thrown during the call to `T` 's copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T` 's copy assignment.

This operator shall be defined as deleted unless `T` is `void` or `is_copy_assignable_v<T>` and `is_copy_assignable_v<E>` and `is_copy_constructible_v<E>` and `is_copy_constructible_v<T>` and `is_nothrow_move_constructible_v<E>` .

```
expected<T, E>& operator=(expected<T, E>&& rhs) noexcept(/*see below*/);
```

Effects:

If `*this` contains a value and `rhs` contains a value, move assign `*rhs` to the contained value `val` if `T` is not `void` ;

otherwise, if `*this` does not contain a value and `rhs` does not contain a value, move assign `unexpected(rhs.error())` to the contained value `unexpect` ;

otherwise, if `*this` contains a value and `rhs` does not contain a value,

- destroys the contained value by calling `val.~T()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with `unexpected(move(forward<expected<T, E>>(rhs)))` ;

otherwise `*this` does not contain a value and `rhs` contains a value

if `is_nothrow_move_constructible_v<T>`

- destroys the contained value by calling `unexpect.~unexpected<E>()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `*move(rhs)` ;

otherwise as `is_nothrow_move_constructible_v<E>`

- move constructs a new `unpepected_type<E> tmp` on the stack from `unexpected(this->error())` (which can't throw as `E` is nothrow-move-constructible),

- destroys the contained value by calling `unexpect.~unexpected<E>()`,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `*move(rhs)`. Either,
  - The constructor didn't throw, so mark the expected as holding a `T` (which can't throw), or
  - The constructor did throw, so move-construct the `unexpected<E>` from the stack `tmp` back into the expected storage (which can't throw as `E` is nothrow-move-constructible), and rethrow the exception.

Returns: `*this`.

Postconditions: `bool(rhs) == bool(*this)`.

Remarks: The expression inside noexcept is equivalent to: `is_nothrow_move_assignable_v<T> && is_nothrow_move_constructible_v<T>`.

If any exception is thrown, the values of `bool(*this) and bool(rhs)` remain unchanged. If an exception is thrown during the call to `T`'s copy constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment. If an exception is thrown during the call to `E`'s copy assignment, the state of its contained unexpected value is as defined by the exception safety guarantee of `E`'s copy assignment.

This operator shall be defined as deleted unless `is_move_constructible_v<T>` and `is_move_assignable_v<T>` and `is_nothrow_move_constructible_v<E>` and `is_nothrow_move_assignable_v<E>`.

```
template <class U>
  expected<T, E>& operator=(U&& v);
```

Effects:

If `*this` contains a value, assigns `forward<U>(v)` to the contained value `val`;

otherwise, if `is_nothrow_constructible_v<T, U&&>`

- destroys the contained value by calling `unexpect.~unexpected<E>()`,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `forward<U>(v)` and
- set `has_value` to `true`;

otherwise as `is_nothrow_constructible_v<E, U&&>`

- move constructs a new `unexpected<E> tmp` on the stack from `unexpected(this->error())` (which can't throw as `E` is nothrow-move-constructible),
- destroys the contained value by calling `unexpect.~unexpected<E>()`,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `forward<U>(v)`. Either,
  - the constructor didn't throw, so mark the expected as holding a `T` (which can't throw), that is set `has_val` to `true`, or
  - the constructor did throw, so move construct the `unexpected<E>` from the stack `tmp` back into the expected storage (which can't throw as `E` is nothrow-move-constructible), and re-throw the exception.

Returns: `*this`.

Postconditions: `*this` contains a value.

Remarks: If any exception is thrown, the value of `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment.

This function shall not participate in overload resolution unless: - `is_same_v<expected<T,E>, decay_t<U>>` is `false` and - `conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>>` is `false`, - `is_constructible_v<T, U>` is `true`, - `is_assignable_v<T&, U>` is `true` and - `is_nothrow_move_constructible_v<E>` is `true`.

```
expected<T, E>& operator=(unexpected<E> const& e) noexcept(`see below`);
```

Effects:

If `*this` does not contain a value, assigns `unexpected(rhs.error())` to the contained value `unexpect`;

otherwise,

- destroys the contained value by calling `val.~T()`,
- initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with `unexpected(forward<expected<T, E>>(rhs))` and set `has_val` to `false`.

Returns: `*this`.

Postconditions: `*this` does not contain a value.

Remarks: If any exception is thrown, value of valued remains unchanged.

This signature shall not participate in overload resolution unless `is_nothrow_copy_constructible_v<E>` and `is_assignable_v<E&, E>`.

```
expected<T, E>& operator=(unexpected<E> && e);
```

Effects:

If `*this` does not contain a value, move assign `unexpected(rhs.error())` to the contained value `unexpect` ;

otherwise,

- destroys the contained value by calling `val.~T()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `unexpected<E>` with `unexpected(move(forward<expected<T, E>>(rhs)))` and set `has_val` to `false` .

Returns: `*this` .

Postconditions: `*this` does not contain a value.

Remarks: If any exception is thrown, value of valued remains unchanged.

This signature shall not participate in overload resolution unless `is_nothrow_move_constructible_v<E>` and `is_move_assignable_v<E&, E>` .

```
template <class... Args>
  void emplace(Args&&... args);
```

Effects:

If `*this` contains a value, assigns `forward<U>(v)` to the contained value `val` as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...`

otherwise, if `is_nothrow_constructible_v<T, Args&&...>`

- destroys the contained value by calling `unexpect.~unexpected<E>()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `std::forward<Args>(args)...` and
- set `has_value` to `true` ;

otherwise as `is_nothrow_constructible_v<T, U&&>`

- move constructs a new `unexpected<E> tmp` on the stack from `unexpected(this->error())` (which can't throw as `E` is nothrow-move-constructible),
- destroys the contained value by calling `unexpect.~unexpected<E>()` ,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `forward<U>(v)` . Either,
  - the constructor didn't throw, so mark the expected as holding a `T` (which can't throw), that is set `has_value` to `true` , or
  - the constructor did throw, so move-construct the `unexpected<E>` from the stack `tmp` back into the expected storage (which can't throw as `E` is nothrow-move-constructible), and re-throw the exception.

if `*this` contains a value, assigns the contained value `val` as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...` ; otherwise, destroys the contained value by calling `unexpect.~unexpected<E>()` and initializes the contained value as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...` .

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected assignment of `T` .

Remarks: If an exception is thrown during the call to `T` 's assignment, nothing changes.

This signature shall not participate in overload resolution unless `is_no_throw_constructible_v<T, Args&&...>` .

```
template <class U, class... Args>
  void emplace(initializer_list<U> il, Args&&... args);
```

Effects: if `*this` contains a value, assigns the contained value `val` as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...` , otherwise destroys the contained value by calling `unexpect.~unexpected<E>()` and initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...` .

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected assignment of `T` .

Remarks: If an exception is thrown during the call to `T` 's assignment nothing changes.

The function shall not participate in overload resolution unless: `is_no_throw_constructible_v<T, initializer_list<U>&, Args&&...>` .

X.Z.4.4 Swap [expected.object.swap]

```
void swap(expected<T, E>& rhs) noexcept(/*see below*/);
```

Effects: if `*this` contains a value and `rhs` contains a value, calls `swap(val, rhs.val)`, otherwise if `*this` does not contain a value and `rhs` does not contain a value, calls `swap(err, rhs.err)`, otherwise exchanges values of `rhs` and `*this`.

Throws: Any exceptions that the expressions in the Effects clause throw.

Remarks: **TODO**

Remarks: The expression inside noexcept is equivalent to: `is_nothrow_move_constructible_v<T> and noexcept(swap(declval<T&>(), declval<T&>())) and is_nothrow_move_constructible_v<E> and noexcept(`
The function shall not participate in overload resolution unless: LValues of type `T` shall be `Swappable`, `is_move_constructible_v<T>`, LValues of type `E` shall be `Swappable` and `is_move_constructible_v<T>`.

X.Z.4.5 Observers [expected.object.observe]

```
constexpr const T* operator->() const;
T* operator->();
```

Requires: `*this` contains a value.

Returns: `&val`.

Remarks: Unless `T` is a user-defined type with overloaded unary `operator&`, the first function shall be a constexpr function.

```
constexpr const T& operator *() const&;
T& operator *() &;
```

Requires: `*this` contains a value.

Returns: `val`.

Remarks: The first function shall be a constexpr function.

```
constexpr T&& operator *() &&;
constexpr const T&& operator *() const&&;
```

Requires: `*this` contains a value.

Returns: move(val).

Remarks: This function shall be a constexpr function.

```
constexpr explicit operator bool() noexcept;
```

Returns: `has_val`.

Remarks: This function shall be a constexpr function.

```
constexpr void expected<void, E>::value() const;
```

Throws:

- `bad_expected_access(err)` if `*this` does not contain a value.

```
constexpr const T& expected::value() const&;
constexpr T& expected::value() &;
```

Returns: `val`, if `*this` contains a value.

Throws:

- Otherwise `bad_expected_access(err)` if `*this` does not contain a value.

Remarks: These functions shall be constexpr functions.

```
constexpr T&& expected::value() &&;
constexpr const T&& expected::value() const&&;
```

Returns: `move(val)`, if `*this` contains a value.

Throws:

- Otherwise `bad_expected_access(err)` if `*this` does not contain a value.

Remarks: These functions shall be constexpr functions.

```
constexpr const E& error() const&;
constexpr E& error() &;
```

Requires: `*this` does not contain a value.

Returns: `unexpect.value()`.

Remarks: The first function shall be a constexpr function.

```
constexpr E&& error() &&;
constexpr const E&& error() const &&;
```

Requires: `*this` does not contain a value.

Returns: `move(unexpect.value())`.

Remarks: The first function shall be a constexpr function.

```
template <class U>
  constexpr T value_or(U&& v) const&;
```

Effects: Equivalent to `return bool(*this) ? **this : static_cast<T>(std::forward<U>(v));`.

Remarks: If `is_copy_constructible_v<T>` and `is_convertible_v<U&&, T>` is false the program is ill-formed.

```
template <class U>
  T value_or(U&& v) &&;
```

Effects: Equivalent to `return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));`.

Remarks: If `is_move_constructible_v<T>` and `is_convertible_v<U&&, T>` is false the program is ill-formed.

X.Z.6 `unexpect` tag [expected.unexpect]

```
struct unexpect_t {
    explicit unexpect_t() = default;
};
inline constexpr unexpect_t unexpect{};
```

X.Z.7 Template Class `bad_expected_access` [expected.badexpectedaccess]

```
template <class E>
class bad_expected_access : public bad_expected_access<void> {
public:
    explicit bad_expected_access(E);
    virtual const char* what() const noexcept overrride;
    const E& error() const&;
    E& error() &;
    E&& error() &&;
private:
    E val; // exposition only
};
```

The template class `bad_expected_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of a unexpected expected object.

```
bad_expected_access::bad_expected_access(E e);
```

Effects: Constructs an object of class `bad_expected_access` storing the parameter.

Postconditions: `what()` returns an implementation-defined NTBS.

````
    const E& error() const&;
    E& error() &;
 ```

*Effects*: Equivalent to: `return val;`


##############################################################################
```c++
    E&& error() &&;
    const E&& error() const &&;
 ```

*Effects*: Equivalent to: `return move(val);`


##############################################################################
```c++
virtual const char* what() const noexcept overrride;
````

Returns: An implementation-defined NTBS.

X.Z.7 Template Class `bad_expected_access<void>` [expected.badexpectedaccess_base]

```
template <>
class bad_expected_access<void> : public exception {
public:
    explicit bad_expected_access();
};
```

The template class `bad_expected_access<void>` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of a unexpected expected object.

X.Z.8 Expected Relational operators [expected.relational_op]

```
    template <class T, class E>
        constexpr bool operator==(const expected<T, E>& x, const expected<T, E>& y);
```

Requires: `T` (if not `void`) and `unexpected<E>` shall meet the requirements of *EqualityComparable*.

Returns: If `bool(x) != bool(y)`, `false`; otherwise if `bool(x) == false`, `unexpected(x.error()) == unexpected(y.error())`; otherwise `true` if `T` is `void` or `*x == *y` otherwise.

Remarks: Specializations of this function template, for which `T` is `void` or `*x == *y` and `unexpected(x.error()) == unexpected(y.error())` are core constant expression, shall be constexpr functions.

```
    template <class T, class E>
        constexpr bool operator!=(const expected<T, E>& x, const expected<T, E>& y);
```

Requires: `T` (if not `void`) and `unexpected<E>` shall meet the requirements of *EqualityComparable*.

Returns: If `bool(x) != bool(y)`, `true`; otherwise if `bool(x) == false`, `unexpected(x.error()) != unexpected(y.error())`; otherwise `true` if `T` is `void` or `*x != *y`.

Remarks: Specializations of this function template, for which `T` is `void` or `*x != *y` and `unexpected(x.error()) != unexpected(y.error())` are core constant expression, shall be constexpr functions.

X.Z.9 Comparison with `T` [expected.comparison_T]

```
    template <class T, class E> constexpr bool operator==(const expected<T, E>& x, const T& v);
    template <class T, class E> constexpr bool operator==(const T& v, const expected<T, E>& x);
```

Requires: `T` is not `void` and the expression `*x == v` shall be well-formed and its result shall be convertible to `bool`. [Note: `T` need not be *EqualityComparable*. - end note]

Effects: Equivalent to: `return bool(x) ? *x == v : false;`.

```
    template <class T, class E> constexpr bool operator!=(const expected<T, E>& x, const T& v);
    template <class T, class E> constexpr bool operator!=(const T& v, const expected<T, E>& x);
```

Requires: `T` is not `void` and the expression `*x == v` shall be well-formed and its result shall be convertible to `bool`. [Note: `T` need not be *EqualityComparable*. - end note]

Effects: Equivalent to: `return bool(x) ? *x != v : false;`.

X.Z.10 Comparison with unexpected<E> [expected.comparisonunexpectedE]

```
template <class T, class E> constexpr bool operator==(const expected<T, E>& x, const unexpected<E>& e);
template <class T, class E> constexpr bool operator==(const unexpected<E>& e, const expected<T, E>& x);
```

Requires: The expression `unexpected(x.error()) == e` shall be well-formed and its result shall be convertible to `bool`. [Note: `E` need not be *EqualityComparable*. - end note]

Effects: Equivalent to: `return bool(x) ? true : unexpected(x.error()) == e;`.

```
template <class T, class E> constexpr bool operator!=(const expected<T, E>& x, const unexpected<E>& e);
template <class T, class E> constexpr bool operator!=(const unexpected<E>& e, const expected<T, E>& x);
```

Requires: The expression `unexpected(x.error()) != e` shall be well-formed and its result shall be convertible to `bool`. [Note: `E` need not be *EqualityComparable*. - end note]

Effects: Equivalent to: `return bool(x) ? false : unexpected(x.error()) != e;`.

X.Z.11 Specialized algorithms [expected.specalg]

```
template <class T, class E>
void swap(expected<T, E>& x, expected<T, E>& y) noexcept(noexcept(x.swap(y)));
```

Effects: Calls `x.swap(y)`.

Remarks: This function shall not participate in overload resolution unless `T is` void `or` ismoveconstructiblev `is` true `,` isswappablev `is` true `and` ismoveconstructiblev `is` true `and` isswappablev `is` true`

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++17.

An almost full reference implementation of this proposal can be found at ExpectedImpl.

Acknowledgements

We are very grateful to Andrei Alexandrescu for his talk, which was the origin of this work. We thanks also to every one that has contributed to the Haskell either monad, as either's interface was a source of inspiration.

Thanks to Fernando Cacciola, Andrzej KrzemieÃĔâĂđski and every one that has contributed to the wording and the rationale of N3793.

Thanks to David Sankel, Mark Calabrese, Axel Naumann and those that participated in the Oulu's review for insisting in the extraction of the monadic interface.

Thanks to Niall Douglas for reporting some possible issues in this proposal and for raising alternative design approaches after implementing expected in its Boost.Outcome library. Thank to Andrzej KrzemieÃĔâĂđski and Peter Dimov for all its pertinent exchanges during this review.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- Boost.Outcome Niall Douglas

  https://ned14.github.io/boost.outcome/index.html

- Alexandrescu.Expected A. Alexandrescu. C++ and Beyond 2012 - Systematic Error Handling in C++, 2012.

  http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C

- ERR err - yet another take on C++ error handling, 2015.

  https://github.com/psiha/err

- N3527 Fernando Cacciola and Andrzej Krzemieński. N3527 - A proposal to add a utility class to represent optional objects (Revision 3), March 2013.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3527.html

- N3672 Fernando Cacciola and Andrzej Krzemieński. N3672 - A proposal to add a utility class to represent optional objects (Revision 4), June 2013.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3672.html

- N3793 Fernando Cacciola and Andrzej Krzemieński. N3793 - A proposal to add a utility class to represent optional objects (Revision 5), October 2013.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3793.html

- N3857 H. Sutter S. Mithani N. Gustafsson, A. Laksberg. N3857, improvements to std::future and related apis, 2013.

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3857.pdf

- N4015 Pierre talbot Vicente J. Botet Escriba. N4015, a proposal to add a utility class to represent expected monad, 2014.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4015.pdf

- N4109 Pierre talbot Vicente J. Botet Escriba. N4109, a proposal to add a utility class to represent expected monad (Revision 1), 2014.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4109.pdf

- N4617 N4617 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 DTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf

- P0057R2 Wording for Coroutines

  www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0057r2.pdf

- P0088R0 Variant: a type-safe union that is rarely invalid (v5), 2015.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0088r0.pdf

- P0110R0 Implementing the strong guarantee for `variant<>` assignment

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0110r0.html

- P0157R0 L. Crowl. P0157R0, Handling Disappointment in C++, 2015.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html

- P0159R0 Artur Laksberg. P0159R0, Draft of Technical Specification for C++ Extensions for Concurrency, 2015.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html

- P0262R0 L. Crowl, C. Mysen. A Class for Status and Optional Value.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0262r0.html

- P0323R0 A proposal to add a utility class to represent expected monad (Revision 2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0323r0.pdf

- P0323R1 A proposal to add a utility class to represent expected monad (Revision 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0323r1.pdf

- P0323R2 A proposal to add a utility class to represent expected monad (Revision 4)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r2.pdf

- P0393R3 Tony Van Eerd. Making Variant Greater Equal.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0393r3.html

- P0650R0 C++ Monadic interface

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0650r0.html

- P0786R0 *ValuedOrError* and *ValueOrNone* types

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0786r0.html

- ExpectedImpl Vicente J. Botet Escriba. Expected, 2016.

  https://github.com/viboes/std-make/tree/master/include/experimental/fundamental/v3/expected

- a-gotcha-with-optional A gotcha with Optional

  https://akrzemi1.wordpress.com/2014/12/02/a-gotcha-with-optional

History

Revision 5 - Revision of P0323R2 after with feedback from Toronto

The 5th revision of this proposal fixes some typos and takes in account the feedback from Toronto meeting to have wording ready for LWG. JF Bastien co-author this proposal. Next follows the direction of the committee:

- Remove the default value for the `Error` parameter.
- Remove comparison for expected.
- Rename `unexpected_type` and reuse the reserved `unexpected`.
- Remove `make_expected` factory
- Remove `make_unexpected` factory and make use of deduction guides.
- Remove the helper functions and move them to a future *ValueOrError* proposal P0786R0.
- Remove `make_expected_from_call`, `make_expected_from_error`.
- Remove `std::hash` specialization.
- Remove `get_unexpected`.
- `void` is not a valid value for the `Error` parameter.
- Update Open points section.

Other pending changes:

- Remove the Future Work section.
- **TODO** Finish the wording for `expected<void,E>` specialization.
- **TODO** Finish the wording ready for LWG

Other changes proposed by the authors:

- Make `bad_expected_access<E>` inherit from `bad_expected_access<void>`.

Revision 4 - Revision of P0323R1

The 4th revision of this proposal aligns with the late optional changes, complete the wording ensuring the never empty warranties and fixes some typos. In addition it adds more open points concerning whether `expected<T, E>` must be ordered and implicit conversions. Most of the changes come from suggestion done by Niall and from the feedback of the review of the Boost.Outcome library and my understanding of the use cases the Boost.Outcome reveal.

Next follows the main modifications:

- Added rvalue overloads for `bad_expected_access::error()` getters and remove the `constexpr`.
- Provide factories that returns `expected<const T, E>`.
- Adapt from late changes in `optional` concerning the observers.
- `get_unexpected()` returns by reference now.
- Allow construction from `expected<U,G>` when the types are convertible.
- Make `bad_expected_access` inherit from `std::exception` instead of from `std::logic_error.
- Take in account constructor guides.

More open points:

- Consider removing `make_expected_from_call`.
- Consider adding a level on the `bad_expect_access<E>` exception hierarchy.
- Consider changing the default `Error` argument to `error_code`.
- Consider removing the comparison operators and specialize `less<>`.
- Consider `expected<T&, E>`.
- Consider a function that allows to adapt the error transported by `expected`.
- Reconsider expected with a variadic number of errors.

Revision 3 - Revision of P0323R0 after with feedback from Oulu

The 3rd revision of this proposal fixes some typos and takes in account the feedback from Oulu meeting. Next follows the direction of the committee:

- Split the proposal on a simple `expected` class and a generic monadic interface.
- As `variant`, `expected` requires some properties in order to ensure the never-empty warranties. As the error type should be no throw movable, we are always sure to be able to ensure the never-empty warranties (Wording **not yet complete**).
- Removed `exception_ptr` specializations as it introduces different behavior, in particular comparisons, exception thrown, ...
- Adapted comparisons to P0393R3 and consider `T` < `E` to be inline with `variant`.
- Redefined the meaning of `e = {}` as `expected<T, E>` defaults to `T()`.
- Added `const &&` overloads for value getters.
- Consider to adapt the constructor and assignment from convertible to `T` and `E` to follow last changes in `std::optional` (Wording **not yet complete**).
- Considered making the conversion from the value type explicit and remove the mixed operations to make the interface more robust even if less friendly.

- Removed the future work section.

Revision 2 - Revision of N4109 after discussion on the ML

- Fix default constructor to `T`. N4109 should change the default constructor to `T`, but there were some inconsistencies.
- Complete wording comparison.
- Adapted to last version of referenced proposals.
- Moved alternative designs from open questions to an Appendix.
- Moved already answered open points to a Rationale section.
- Moved open points that can be decided later to a future directions section.
- Complete wording hash.
- Add a section for adapting to `await`.
- Add a section in future work about a possible variadic.
- Fix minor typos.

Not done yet

- As `variant`, `expected` requires some properties in order to ensure the never-empty warranties. Add more on never-empty warranties and replace the wording.

Revision 1 - Revision of N4015 after Rapperswil feedback:

- Switch the expected class template parameter order from `expected<E,T>` to `expected<T, E>`.
- Make the unexpected value a salient attribute of the expected class concerning the relational operators.
- Removed open point about making `expected<T, E>` and `expected<T>` different classes.

Appendix I: Alternative designs

A Configurable Expected

Expected might be configurable through a trait `expected_traits`.

The first variation point is the behavior of `value()` when `expected<T, E>` contains an error. The current strategy throw a `bad_expected_access` exception but it might not be satisfactory for every error type. For example, some might want to encapsulate an `error_code` into a `system_error`. Or in debug mode, they might want to use an `assert` call.

We could as well make the exception thrown depend on the Error overloading a `rethrow_on_unexpected`.

Which exception throw when the user try to get the expected value but there is none?

It has been suggested to let the user decide the exception that would be throw when the user try to get the expected value but there is none, as third parameter.

While there is no major complexity doing it, as it just needs a third parameter that could default to the appropriated class,

```
template <class T, class Error, class Exception = bad_expected_access>
struct expected;
```

The authors consider that this is not really needed and that this parameter should not really be part of the type.

The user could use `value_or_throw()`

```
expected<int, std::error_code> f();
expected<int, std::error_code> e = f();
auto i = value_or_throw<std::system_error>(e);
```

where

```
template <class Exception, class T, class E>
constexpr const T& value_or_throw(expected<T, E> const& e)
{
    if (!e.has_value())
       throw Exception(e.error());
    return *e;
}
```

A function like this one could be added to the standard, but this proposal doesn't request it.

An alternative is to overload the `value` function with the exception to throw.

```
template <class Exception, class T, class E>
constexpr value_type const& value() const&
```

About `expected<T, ErrorCode, Exception>`

It has been suggested also to extend the design into something that contains

- a `T`, or
- an `ErrorCode`, or
- an `exception_ptr`

This is the case of [Outcome] library.

Again there is no major difficulty to implement it, but instead of having one variation point we have two, that is, is there a value, and if not, if is there an `exception_ptr`.

Better to have a variadic `expected<T, E...>`

Appendix II: Related types

Variant

`expected<T, E>` can be seen as a specialization of `boost::variant<unexpected<E>,T>` which gives a specific intent to its first parameter, that is, it represents the type of the expected contained value. This specificity allows to provide a pointer like interface, as it is the case for `std::optional<T>`. Even if the standard included a class `variant<T, E>`, the interface provided by `expected<T, E>` is more specific and closer to what the user could expect as the result type of a function. In addition, `expected<T, E>` doesn't intend to be used to define recursive data as `boost::variant<>` does.

The following table presents a brief comparison between `boost::variant<unexpected<E>, T>` and `expected<T, E>`.

	std::variant<T, unexpected<E>>	**expected<T, E>**
never-empty warranty	no	yes
accepts is_same<T, E>	yes	yes
swap	yes	yes
factories	no	expected / unexpected
hash	no	yes
value_type	no	yes
default constructor	yes (if T is default constructible)	yes (if T is default constructible)
observers	boost::get<T> and boost::get<E>	pointer-like / value / error / value_or
visitation	visit	no

Optional

We can see `expected<T, E>` as an `std::optional<T>` that collapse all the values of `E` to `nullopt`.

We can convert an `expected<T, E>` to an `optional<T>` with the possible loss of information.

```
template <class T>
optional<T> make_optional(expected<T, E> v) {
    if (v) return make_optional(*v);
    else nullopt;
}
```

We can convert an `optional<T>` to an `expected<T, E>` without knowledge of the root cause. We consider that `E()` is equal to `nullopt` since it shouldn't bring more informations (however it depends on the underlying error — we considered `exception_ptr` and `error_code`).

```
template <class E, class T>
expected<T, E> make_expected(optional<T> v) {
    if (v) return *v;
    else unexpected(E());
}
```

The problem is if `E` is a status and `E()` denotes a success value.

Promise and Future

We can see `expected<T>` as an always ready `future<T>`. While `promise<>` / `future<>` focuses on inter-thread asynchronous communication, `excepted<T, E>` focus on eager and synchronous computations. We can move a ready `future<T>` to an `expected<T>` with no loss of information.

```
template <class T>
expected<T, exception_ptr> make_expected(future<T>&& f) {
    assert (f.ready() && "future not ready");
    try {
        return f.get();
    } catch (...) {
        return unexpected<exception_ptr>{current_exception()};
    }
}
```

We could also create a `future<T>` from an `expected<T>`

```
template <class T>
future<T> make_future(expected<T> e) {
    if (e)
        return make_ready_future(*e);
    else
        return make_exceptional_future<T>(e.error());
};
```

Comparison between optional, expected and future

The following table presents a brief comparison between `optional<T>`, `expected<T, E>` and `promise<T>` / `future<T>`.

	optional	expected	promise/future
specific null value	yes	no	non
relational operators	yes	yes	no
swap	yes	yes	yes
factories	make_optional / nullopt	expected / unexpected	make_ready_future / make_exceptional_future
hash	yes	no	yes
value_type	yes	yes	no
default constructor	yes	yes (if T is default constructible)	yes
allocators	no	no	yes
emplace	yes	yes	no
bool conversion	yes	yes	no
state	bool()	bool() / valid	valid / ready
observers	pointer-like / value / value_or	pointer-like / value / error / value_or	get
visitation	no	no	then
grouping	n/a	n/a	when_all / when_any