Document number:	P0323R1
Date:	2016-10-12
Revises:	N4109/N4015
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

A proposal to add a utility class to represent expected object (Revision 3)

Abstract

Class template `expected<T,E>` proposed here is a type that may contain a value of type `T` or a value of type `E` in its storage space. `T` represents the expected value, `E` represents the reason explaining why it doesn't contains a value of type `T`. The interface and the rational are based on `std::optional` N3793. We can consider `expected<T,E>` as a generalization of `optional<T>`.

The monadic interface has been removed from this revision, and so the title has been changed.

Table of Contents

History

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>` default to `T()`.
- Added `const &&` overloads for value getters.
- Consider the 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 and expected different classes.

Introduction

Class template `expected<T,E>` proposed here is a type that may contain a value of type `T` or a value of type `E` in its storage space. `T` represents the expected value, `E` represents the reason explaining why it doesn't contains a value of type `T`, that is, the unexpected value. Its interface allows to query if the underlying value is either 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++ talk Alexandrescu.Expected. The interface and the rational are based on `std::optional` N3793. We can consider that `expected<T,E>` is a generalization of `optional<T>` providing in addition some specific functions associated to the unexpected type `E`. It requires no changes to core language, and breaks no existing code.

There is a related proposal for a class including a status and an optional value P0262R0. P0157R0 describes when to use each of the different error report mechanism.

Motivation

Basically, the two main error mechanisms are exceptions and return codes. Before further explanation, we should ask us what are the characteristics of a good error mechanism.

- **Error visibility**: Failure cases should appear throughout the code review. Because the debug can be painful if the errors are hidden.
- **Information on errors**: The errors should carry out as most as possible information from their origin, causes and possibly the ways to resolve it.
- **Clean code**: The treatment of errors should be in a separate layer of code and as much invisible as possible. So the code reader could notice the presence of exceptional cases without stop his reading.
- **Non-Intrusive error** The errors should not monopolize a communication channel dedicated to the 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 to be quite contradictory and deserve further explanation. The former points out that errors not handled should appear clearly in the code. The latter tells us that the error handling mustn't interfere with the code reading, meaning that it clearly shows the normal execution flow. A comparison between the exception and return codes is given in the next table.

	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 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.

- **Error visibility**: It takes the best of the exception and error code. It's visible because the return type is `expected<T,E>` and the user cannot ignore the error case if he wants to retrieve the contained value.
- **Information**: Arbitrarily rich.
- **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.
- **Non-Intrusive** Use the return channel without monopolizing it.

It worths mentioning the other characteristics of `expected<T,E>`:

- Associates errors with computational goals.
- Naturally allows multiple errors inflight.
- Teleportation possible.
- Across thread boundaries.
- Across no-throw subsystem boundaries.
- Across time: save now, throw later.
- Collect, group, combine errors.

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::error_condition` 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 (the boilerplate code concerning `std::error_condition` is not shown):

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

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

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

(1) The implicit conversion from `unexpected_type<E>` to `expected<T,E>` and (2) from `T` to `expected<T,E>` prevents using too much boilerplate code. The advantages are that 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, error_condition>`:

```
expected<double, error_condition> 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 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, error_condition> safe_divide(int i, int j)
{
    if (j == 0) return make_unexpected(arithmetic_errc::divide_by_zero);
    if (i%j != 0) return make_unexpected(arithmetic_errc::not_integer_division);
    else return i / j;
}
```

Error retrieval and correction

The major advantage of `expected<T,E>` over `optional<T>` is the ability to transport an error, but we didn't come yet to an example that retrieve the error. First of all, we should wonder what a programmer do when a function call returns an error:

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

3. Trying to resolve the error.

Because the first behavior might lead to buggy application, we won't consider it in a first time. The handling is dependent of the underlying error type, we consider the `exception_ptr` and the `error_condition` types.

We spoke about how to use the value contained in the `expected` but didn't discuss yet the error usage.

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` :

```
expected<int, error_condition> divide2(int i, int j)
{
    auto e = safe_divide(i,j);
    if (!e && e.error().value() == arithmetic_errc::divide_by_zero) {
        return 0;
    }
    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 as `optional<T>` with some exceptions. That is, we see `expected<T,E>` as `optional<T>` for which all the values of `E` collapse into a single value `nullopt` . In the following sections we present the specificities of the rationale in N3672 applied to `expected<T,E>` .

Conceptual model of expected

`expected<T,E>` models a discriminated union of types `T` and `unexpected_type<E>` . `expected<T,E>` is viewed as a value of type `T` or value of type `unexpected_type<E>` , allocated in the same storage, along with the way of determining 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 `make_unexpected(E)` function is proposed for this purpose. However `T` cannot be `unexpected_type<E>` for a given `E` .

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

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

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

Initialization of `expected<T,E>`

In cases `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 (as `expected<T>` should behave as `T` as much as possible).

```
string s"STR";

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

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

In order to create an unexpected object, the special function `make_unexpected` needs to be used:

```
expected<string, int> ep{make_unexpected(-1)}; // unexpected value, requires Movable<E>
expected<string, int> eq = make_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, error_condition> eg; // expected value
expected<MoveOnly, error_condition> eh{}; // expected value
expected<MoveOnly, error_condition> ei{in_place}; // calls MoveOnly{} in place
expected<MoveOnly, error_condition> 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 a `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`.

Almost never-empty guaranty

As `boost::variant<T,unexpected_type<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 some requirements as described in P0110R0. Given the nature of the parameter `E`, that is, to transport an error, it is expected that `is_nothrow_copy_constructible<E>`, `is_nothrow_move_constructible<E>`, `is_nothrow_copy_assignable<E>` and `is_nothrow_move_assinable<E>`.

If `is_nothrow_constructible<T, Args...>` is `false` `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>`.

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

- Should the default constructor of `expected<T,E>` behave like `variant<T, unexpected_type<E>>` or as `variant<unexpected_type<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 `make_unexpected(E())`? This would be equivalent to the initialization of `variant<unexpected_type<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>`, however as the committee as requested it the paper proposes that `expected<T,E>` default constructor should behave as constructed with `T()` if `T` is default constructible.

Conversion from `T`

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

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

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

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

If the latter appears too cumbersome, one can always use function `make_expected` described below:

```
expected<int> ei = make_expected(1);
auto ej = make_expected(1);
```

or simply using deduced template parameter for constructors

```
expected<int> ei = expected(1);
auto ej = expected(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 `expected_type<T>` (similar to `unexpected_type<E>` explicitly convertible from `T` and implicitly convertible to `expected<T,E>`.

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 a special non-member `make_unexpected` function to indicate an unexpected state for `expected<T,E>`. It is used for construction and assignment. This might rise 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, int> exp1 = make_unexpected(1);
expected<string, int> exp2 = {unexpect, 1};
exp1 = make_unexpected(1);
exp2 = {unexpect, 1};
```

or simply using deduced template parameter for constructors

```
expected<string, int> exp1 = unexpected_type(1);
exp1 = unexpected_type(1);
```

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

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

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

Should we support the `exp2 = {}`?

Note also that the definition of `unexpected_type` 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 doesn't 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>` and so, the following example combines initialization and value-checking in a boolean context.

```
if (expected<char, error_condition> 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 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

It was chosen to use indirection operator 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 has risen 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 a object that doesn't contain a value is an 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 depends on the expected error type. By default it throws `bad_expected_access<E>` (derived from `std::logic_error`) which will contain the stored error.

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

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 `! bool(*this)`.

```
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`.

Conversion to the unexpected value

As the `error()` function, the `get_unexpected()` works only if the expected object has no value. It is used to propagate errors. Note that the following equivalences yield:

```
f.get_unexpected() == make_unexpected(f.error());
f.get_unexpected() == expected<T, E>{unexpect, f.error()};
```

This member is provided for convenience, it is further demonstrated in the next example:

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

`get_unexpected` is also provided for symmetry purpose. On one side, there is an implicit conversion from `unexpected_type<E>` to `expected<T,E>` and on the other side there is an explicit conversion from `expected<T,E>` to `unexpected_type<E>`.

Function `value_or`

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

Relational operators

As `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.

Ordering is essential if we want to store expected values in ordered associative containers. A number of ways of including the unexpected state in comparisons have been suggested. The ones proposed, follows P0393R3: `unexpected` values stored in `expected<T,E>` are simply treated as additional values that are always different from `T`; these values are always compared as greater than any value of `T` when stored in an `expected` object. This is because we see `expected<T,E>` as a specialization of `variant<T, unexpected_type<E>`

But how to define the relational operators for `unexpected_type<E>`? We can forward the request to the respective `E` relational operators when `E` defines these operators, don't support the operators if `unexpected_type<E>` doesn't define `operator<()` `operator<()`.

This limitation is one of the main motivations for having a user defined type `E` with strict weak ordering. E.g. if the user know the exact types of the exceptions that can be thrown `E1`, ..., `En`, the error parameter could be some kind of `std::variant<E1, ... En>` for which a strict weak ordering can be defined. If the user would like to take care of unknown exceptions something like `std::variant<std::monostate, E1, ... En>` would be a quite appropriated model.

```
expected<unsigned, int> e0{0};
expected<unsigned, int> e1{1};
expected<unsigned, int> eN{unexpect, -1};
assert (eN > e0);
assert (e0 > e1);
assert (!(eN > eN));
assert (!(e1 > e1))
assert (eN != e0);
assert (e0 != e1);
assert (eN == eN);
assert (e0 == e0);
```

Given that both `unexpected_type<E>` and `T` are implicitly convertible to `expected<T,E>`, this implies the existence and semantics of mixed comparison between `expected<T,E>` and T `, as well as between `expected `and` unexpected_type`:

```
assert (eN == make_unexpected(1));
assert (e0 != make_unexpected(1));
assert (eN != 1);
assert (e1 == 1);
assert (eN > 1);
assert (e0 > make_unexpected(1));
```

Although it is difficult to imagine any practical use case of ordering relation between `expected<T,E>` and `unexpected_type<E>`, we still provide it for completeness sake as we have for `optional` and becaus ethe operation can be implemented more efficiently. Note however that `std::variant` doesn't defines these mixed operations.

The mixed relational operators, especially those representing order, between `expected<T,E>` and `T` have been accused of being dangerous. In code examples like the following, it may be unclear if the author did not really intend to compare two `T` 's.

```
auto count = get_expected_count();
if (count < 20) {} // or did you mean: *count < 20 ?
if (! count || *count < 20) {} // verbose, but unambiguous
```

Given that `expected<T,E>` is comparable and implicitly constructible from `T`, the mixed comparison is there already. We would have to artificially create the mixed overloads only for them to cause controlled compilation errors. A consistent approach to prohibiting mixed relational operators would be to also prohibit the conversion from T or to also prohibit homogenous relational operators for `expected<T,E>` ; we do not want to do either, for other reasons discussed in this proposal. Also, mixed relational operations are available in `std::optional<T>`. Note however that `expected<T,nullopt_t>` and `optional<T>` behave the opposite. This is a consequence of having reverted the `T` and `E` parameters and so defaulting to `T()`.

Modifiers

Reseting 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 warranties, `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 neither.

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.

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>` do not modify the state of the source (valued or erroneous) of `expected` and the move semantics is up to `T` or `E`.

IO operations

For the same reasons than `optional` N3672 we do not add `operator<<` and `operator>>` IO operations.

What happens when `E` is a status ?

When `E` is a status as most of the error codes do 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, id there is only one value `e` that mean success, there is no such need and `expected<T,E>` compose better with the monadic interface.

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

It has been argued that the error should be always available and that often there is a success value associated to the error.

`expected<T,E>` would be seen more like something like `struct {E; optional<T>}`.

The following code show a use case

```
auto e = function();
switch (e.status())
    success: ....; break;
    too_green: ....; break;
    too_pink: ....; break;
```

With the current interface the user could be tempted to do

```
auto e = function();
if (e)
    /*success:*/ ....;
else
    switch (e.error())
    case too_green: ....; break;
    case too_pink: ....; break;
```

This could be done with the current interface as follows

```
auto e = function();
switch (error_or(e, success))
    success: ....; break;
    too_green: ....; break;
    too_pink: ....; break;
```

where

```
template <class E, class T>
E error_or(expected<T,E> const&, E err) {
    if (e) return err;
    else return error();
}
```

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

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

Another use case which could look much uglier is if the user had to test for whether or not there was a specific error code.

```
auto e = function();
while ( e.status == timeout ) {
    sleep(delay);
    delay *=2;
    e = function();
}
```

Here we have a value or a hard error. This use case would need to use something like `has_error`

```
e = function();
while ( has_error(e, timeout) )
{
    sleep(delay);
    delay *=2;
    e = function();
}
```

where

```
template <class T, class E>
bool has_error(expected<T,E> const&, E err) {
    if (e) return false;
    else return error()==err;
}
```

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

Open points

Should `expected<T,E>` be explicitly convertible from `T` and implicitly convertible from `expected_type<T>` ?

Should `expected<T,E>` 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>` ?

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

Should `get_unexpected()` return by reference?

The current implementation stores E instead of unexpected_type, and so we are unable to return `get_unexpected()` by reference. As we see `expected<T,E>` as `variant<T,unexpected_type<E>` we should provide a reference access to `unexpected_type<E>` ?

In addition, if we want to see as a sum type of `T` and `unexpected_type<E>` , we would need this access.

Proposed Wording

The proposed changes are expressed as edits to N4564 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_type` that wraps objects intended as unexpected. This wrapped unexpected object is used to be 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_type;

    // X.Y.4, Unexpected factories
    template <class E>
      constexpr unexpected_type<decay_t<E>> make_unexpected(E&& v);

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

}}}
```

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

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

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

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

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

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

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

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

Returns: `val` .

X.Y.4 Factories [unexpected.factories]

```
template <class E>
constexpr unexpected_type<decay_t<E>> make_unexpected(E&& v);
```

Returns: `unexpected_type<decay_t<E>>(v)`.

X.Y.5 Unexpected Relational operators [unexpectedtype.relationalop]

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

Requires: `x.value() < y.value()`.

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_type<E>& x, const unexpected_type<E>& y);
```

Requires: `x.value() > y.value()`.

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_type<E>& x, const unexpected_type<E>& y);
```

Requires: `x.value() <= y.value()`.

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_type<E>& x, const unexpected_type<E>& y);
```

Returns: `!(x < y)`.

-- 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= error_condition>
        class expected;

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

    // X.Z.6, unexpect tag
    struct unexpect_t{
       unexpect_t() = delete;
    };
    constexpr unexpect_t unexpect{'implementation defined'};

    // X.Z.7, class bad_expected_access
    class bad_expected_access;

    // X.Z.8, 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>&);
    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>&);
    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.9, 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>&);
    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>&);
    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_type<E>
    template <class T, class E>
      constexpr bool operator==(const expected<T,E>&, const unexpected_type<E>&);
    template <class T, class E>
      constexpr bool operator==(const unexpected_type<E>&, const expected<T,E>&);
    template <class T, class E>
      constexpr bool operator!=(const expected<T,E>&, const unexpected_type<E>&);
    template <class T, class E>
      constexpr bool operator!=(const unexpected_type<E>&, const expected<T,E>&);
    template <class T, class E>
      constexpr bool operator<(const expected<T,E>&, const unexpected_type<E>&);
    template <class T, class E>
      constexpr bool operator<(const unexpected_type<E>&, const expected<T,E>&);
    template <class T, class E>
      constexpr bool operator<=(const expected<T,E>&, const unexpected_type<E>&);
```

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

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

    // X.Z.12, Factories
    template <class T> constexpr expected<decay_t<T>> make_expected(T&& v);
    expected<void> make_expected();
    template <class T, class E>
        constexpr expected<T, decay_t<E>> make_expected_from_error(E&& e);
    template <class T, class E, class U>
        constexpr expected<T, E> make_expected_from_error(U&& u);
    template <class F>
        constexpr expected<typename result_type<F>::type>
    make_expected_from_call(F f);

    // X.Z.13, hash support
    template <class T, class E> struct hash<expected<T,E>>;
    template <class E> struct hash<expected<void,E>>;
}}}
```

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_type<E>` 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;
    template <class U>
      struct rebind {
        using type = expected<U, error_type>;
      };

    // X.Z.4.1, constructors
    constexpr expected() noexcept(see below);
    expected(const expected&);
    expected(expected&&) noexcept(see below);
    constexpr expected(const T&);
    constexpr expected(T&&);
    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&&...);
    constexpr expected(unexpected_type<E> const&);
    template <class Err>
        constexpr expected(unexpected_type<Err> const&);
    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> expected& operator=(U&&);
```

```
        expected& operator=(const unexpected_type<E>&);
    expected& operator=(unexpected_type<E>&&) 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;
    T* operator ->();
    constexpr const T& operator *() const&;
    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&;
    E& error() &;
    constexpr E&& error() &&;
    constexpr const E&& error() const &&;
    constexpr unexpected_type<E> get_unexpected() const;
    template <class U>
        constexpr T value_or(U&&) const&;
    template <class U>
        T value_or(U&&) &&;

private:
    bool has_value; // exposition only
    union
    {
        value_type val; // exposition only
        error_type err; // exposition only
    };
};
```

Valued instances of `expected<T,E>` where `T` and `E` is of object type shall contain a value of type `T` or a value of type `E` within its own storage. This value is 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 `E`. Members `has_value`, `val` and `err` are provided for exposition only. Implementations need not provide those members. `has_value` indicates whether the expected object's contained value has been initialized (and not yet destroyed); when `has_value` is true `val` points to the contained value, and when it is false `err` points to the erroneous value.

`T` and `E` shall be an object type and shall satisfy the requirements of `Destructible`.

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

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

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `T{}`.

Postconditions: `bool(*this)`.

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

Remarks: The expression inside noexcept is equivalent to: `is_nothrow_default_constructible<T>::value`.

Remarks: This signature shall not participate in overload resolution unless `is_default_constructible<T>::value`.

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

Effects: If `bool(rhs)` initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.

If `!bool(rhs)` initializes the contained value as if direct-non-list-initializing an object of type `E` with the expression `rhs.error()`.

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

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

Remarks: This signature shall not participate in overload resolution unless `is_copy_constructible<T>::value` and `is_copy_constructible<E>::value`.

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

Effects: If bool(rhs) initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`.

If `!bool(rhs)` initializes the contained value as if direct-non-list-initializing an object of type `E` with the expression `std::move(rhs.error())`.

Postconditions: `bool(rhs) == bool(*this)` and `bool(rhs)` is unchanged.

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

Remarks: The expression inside noexcept is equivalent to: `is_nothrow_move_constructible<T>::value == type and is_nothrow_move_constructible<E>::value`.

Remarks: This signature shall not participate in overload resolution unless `is_move_constructible<T>::value and is_move_constructible<E>::value`.

```
constexpr expected(const T& v);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `v`.

Postconditions: `bool(*this)`.

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.

Remarks: This signature shall not participate in overload resolution unless `is_copy_constructible<T>::value`.

```
constexpr expected(T&& v);
```

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

Postconditions: `bool(*this)`.

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.

Remarks: This signature shall not participate in overload resolution unless `is_move_constructible<T>::value`.

```
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: `bool(*this)`.

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

Remarks: If `T` 's constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

Remarks: This signature shall not participate in overload resolution unless `is_constructible<T, Args&&...>::value`.

```
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: `bool(*this)`.

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

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

If `T` 's constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

Remarks: This signature shall not participate in overload resolution unless `is_constructible<T, initializer_list<U>&, Args&&...>::value`.

```
constexpr expected(unexpected_type<E> const& e);
```

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

Postconditions: `! bool(*this)`.

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

Remark: If `E`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

Remark: This signature shall not participate in overload resolution unless `is_copy_constructible<E>::value`.

```
constexpr expected(unexpected_type<E>&& e);
```

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

Postconditions: `! bool(*this)`.

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

Remark: If `E`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

Remark: This signature shall not participate in overload resolution unless `is_move_constructible<E>::value`.

```
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 `E` with the arguments `std::forward<Args>(args)...`.

Postconditions: `! bool(*this)`.

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

Remarks: If `E`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

Remarks: This signature shall not participate in overload resolution unless `is_constructible<E, Args&&...>::value`.

```
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 `E` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `! bool(*this)`.

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

Remarks: The function shall not participate in overload resolution unless: `is_constructible<E, initializer_list<U>&, Args&&...>::value`.

If `E`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

Remarks: This signature shall not participate in overload resolution unless `is_constructible<E, initializer_list<U>&, Args&&...>::value`.

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

```
~expected();
```

Effects: If `is_trivially_destructible<T>::value != true and bool(*this)`, calls `val->T::~T()`. If `is_trivially_destructible<E>::value != true and ! (bool(*this)`, calls `err->E::~E()`.

Remarks: If `is_trivially_destructible<T>::value and is_trivially_destructible<E>::value` then this destructor shall be a trivial destructor.

X.2.4.3 Assignment [expected.object.assign]

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

Requires: `is_nothrow_copy_constructible<E>::value && is_nothrow_move_constructible<E>::value` is `true`

Effects:

If `! bool(*this) and ! bool(rhs)`, do nothing;

otherwise, if `bool(*this) and bool(rhs)`, assigns `*rhs` to the contained value `val`;

otherwise, if `! bool(*this) and ! bool(rhs)`, assigns `rhs.error()` to the contained value `err`;

otherwise, if `bool(*this) and ! bool(rhs)`,

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

otherwise `! (bool(*this) and bool(rhs)`

if `is_nothrow_copy_constructible<T>::value`

- destroys the contained value by calling `err->E::~E()`
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs`;

otherwise, if `is_nothrow_move_constructible<T>::value`

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

otherwise as `is_nothrow_move_constructible<E>::value`

- move constructs a new `E tmp` on the stack from `this.error()` (which can't throw as `E` is nothrow-move-constructible),
- destroys the contained value by calling `err->E::~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 `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)`.

Exception Safety: 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.

Remarks: This signature shall not participate in overload resolution unless `is_copy_constructible<T>::value and is_copy_assignable<T>::value and is_copy_constructible<E>::value and is_copy_assignable<E>::value`.

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

Effects:

If `! bool(*this) and ! bool(rhs)`, do nothing;

otherwise, if `bool(*this) and bool(rhs)`, assigns `*move(rhs)` to the contained value `val`;

otherwise, if `! bool(*this) and ! bool(rhs)`, assigns `move(rhs).error()` to the contained value `err`;

otherwise, if `bool(*this) and ! bool(rhs)`,

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

otherwise `! (bool(*this) and bool(rhs)`

if `is_nothrow_move_constructible<T>::value`

- destroys the contained value by calling `err->E::~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<E>::value`

- move constructs a new `E tmp` on the stack from `this.error()` (which can't throw as `E` is nothrow-move-constructible),
- destroys the contained value by calling `err->E::~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 `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<T>::value && is_nothrow_move_constructible<T>::value && is_nothrow_move_assignable<E>::value && is_nothrow_mo`

Exception Safety: 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.

Remarks: This signature shall not participate in overload resolution unless
`is_move_constructible<T>::value and is_move_assignable<T>::value and is_move_constructible<E>::value and is_move_assignable<E>::value`.

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

TODO reword this operation to take in account the never empty warranties

Effects:

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

otherwise, if `is_nothrow_move_constructible<T>::value`

- destroys the contained value by calling `err->E::~E()`,
- initializes the contained value as if direct-non-list-initializing an object of type `T` with `forward<U>(v)`;

otherwise as `is_nothrow_move_constructible<E>::value`

- move constructs a new `E tmp` on the stack from `this.error()` (which can't throw as `E` is nothrow-move-constructible),
- destroys the contained value by calling `err->E::~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), or
  - The constructor did throw, so move-construct the `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(*this)`.

Exception Safety: If any exception is thrown, the value of `bool(*this)` 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.

Remarks:

This signature shall not participate in overload resolution unless `is_same<T,U>::value and is_nothrow_constructible<T, Args...>::value`.

[Note: The reason to provide such generic assignment and then constraining it so that effectively `T == U` is to guarantee that assignment of the form `o = {}` is unambiguous. —end note]

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

TODO reword this operation to take in account the never empty warranties

Effects: If `! bool(*this)` assigns `std::forward<E>(e.value())` to the contained value; otherwise destroys the contained value by calling `val->T::~T()` and initializes the contained value as if direct-non-list-initializing object of type `E` with `std::forward<unexpected_type<E>>(e).value()`.

Returns: `*this`.

Postconditions: `! bool(*this)`.

Exception Safety: If any exception is thrown, value of valued remains unchanged. If an exception is thrown during the call to `T` 's constructor, the state of v is determined by exception safety guarantee of `T` 's constructor. If an exception is thrown during the call to `T` 's assignment, the state of `val` and `v` is determined by exception safety guarantee of `T` 's assignment.

Remarks: This signature shall not participate in overload resolution unless `is_copy_constructible<E>::value and is_assignable<E&, E>::value`.

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

Effects: if `bool(*this)`, 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 `err->E::~E()` and initializes the contained value as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...`.

Postconditions: `bool(*this)`.

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

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

Remarks: This signature shall not participate in overload resolution unless `is_no_throw_constructible<T, Args&&...>::value`.

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

Effects: if `bool(*this)`, 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 `err->E::~E()` and initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `bool(*this)`.

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

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

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

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

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

TODO reword this operation to take in account the never empty warranties

Effects: if `bool(*this) and bool(rhs)`, calls `swap(val, rhs.val)`, otherwise if `! bool(*this) and ! bool(rhs)`, calls `swap(err, rhs.err)`, otherwise if `bool(*this) and ! bool(rhs)`, initializes a temporary variable `e` by direct-initialization with `std::move(rhs.err))`, initializes the contained value of rhs by direct-initialization with `std::move(*(*this))`, initializes the expected value of `*this` by direct-initialization with `std::move(rhs.err)` and swaps `has_value` and `rhs.has_value`, otherwise calls to `rhs.swap(*this)`.

Exception Safety: TBC

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

Remarks: The expression inside noexcept is equivalent to: `is_nothrow_move_constructible<T>::value and noexcept(swap(declval<T&>(), declval<T&>())) and is_nothrow_move_constructible<E>::value and`

Remarks: The function shall not participate in overload resolution unless: LValues of type `T` shall be `Swappable`, `is_move_constructible<T>::value`, LValues of type `E` shall be `Swappable` and `is_move_constructible<T>::value`.

X.2.4.5 Observers [expected.object.observe]

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

Requires: `bool(*this)`.

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: `bool(*this)`.

Returns: `val`.

Remarks: The first function shall be a constexpr function.

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

Requires: `bool(*this)` .

Returns: move(val).

Remarks: This function shall be a constexpr function.

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

Returns: `has_value` .

Remarks: This function shall be a constexpr function.

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

Returns: `val` , if `bool(*this)` .

Throws:

• Otherwise `bad_expected_access(err)` if `! bool(*this)` .

Remarks: The first and third functions shall be constexpr functions.

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

Returns: `move(val)` , if `bool(*this)` .

Throws:

• Otherwise `bad_expected_access(err)` if `! bool(*this)` .

Remarks: The first and third functions shall be constexpr functions.

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

Requires: `! bool(*this)` .

Returns: `err` .

Remarks: The first function shall be a constexpr function.

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

Requires: `! bool(*this)` .

Returns: `move(err)` .

Remarks: The first function shall be a constexpr function.

```
constexpr unexpected_type<E> get_unexpected() const;
```

Requires: `! bool(*this).`

Returns: `make_unexpected(err)` .

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

Returns: `bool(*this) ? **this : static_cast<T>(std::forward<U>(v))` .

Exception Safety: If `has_value` and exception is thrown during the call to `T` 's constructor, the value of `has_value` and `v` remains unchanged and the state of `val` is determined by the exception safety guarantee of the selected constructor of `T` . Otherwise, when exception is thrown during the call to `T` 's constructor, the value of *this remains unchanged and the state of `v` is determined by the exception safety guarantee of the selected constructor of `T`.

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

Remarks: If both constructors of `T` which could be selected are constexpr constructors, this function shall be a constexpr function.

Remarks: The function shall not participate in overload resolution unless: `is_copy_constructible<T>::value and is_convertible<U&&, T>::value`.

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

Returns: `bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v))`.

Exception Safety: If `has_value` and exception is thrown during the call to `T`'s constructor, the value of `has_value` and `v` remains unchanged and the state of `val` is determined by the exception safety guarantee of the `T`'s constructor.

Otherwise, when exception is thrown during the call to `T`'s constructor, the value of `*this` remains unchanged and the state of `v` is determined by the exception safety guarantee of the selected constructor of `T`.

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

Remarks: The function shall not participate in overload resolution unless: `is_move_constructible<T>::value and is_convertible<U&&, T>::value`.

X.Z.6 `expected` for `void` [expected.object.void]

```
template <class E>
class expected<void, E>
{
public:
    typedef void value_type;
    typedef E error_type;
    template <class U>
      struct rebind {
        typedef expected<U, error_type> type;
      };
    // ??, constructors
    constexpr expected() noexcept;
    expected(const expected&);
    expected(expected&&) noexcept(see below);
    constexpr explicit expected(in_place_t);
    constexpr expected(unexpected_type<E> const&);
    template <class Err>
      constexpr expected(unexpected_type<Err> const&);
    // ??, destructor
    ~expected();
    // ??, assignment
    expected& operator=(const expected&);
    expected& operator=(expected&&) noexcept(see below);
    void emplace();
    // ??, swap
    void swap(expected&) noexcept(see below);
    // ??, observers
    constexpr explicit operator bool() const noexcept;
    constexpr bool has_value() const noexcept;
    void value() const;
    constexpr const E& error() const&;
    constexpr E& error() &;
    constexpr E&& error() &&;
    constexpr unexpected_type<E> get_unexpected() const;

private:
    bool has_value; // exposition only
    union
    {
        unsigned char dummy; // exposition only
        error_type err; // exposition only
    };
};
```

TODO: Describe the functions

X.Z.7 `unexpect` tag [expected.unexpect]

```
struct unexpect_t;
constexpr unexpect_t unexpect;
```

X.Z.8 Template Class `bad_expected_access` [expected.badexpectedaccess]

```
template <class E>
class bad_expected_access : public logic_error {
public:
    explicit bad_expected_access(E);
    constexpr error_type const& error() const;
    error_type& error();
};
```

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.

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

Returns: The stored error.

Remarks: The first function shall be a constexpr function.

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` and `unexpected_type<E>` shall meet the requirements of *EqualityComparable*.

Returns: If `bool(x) != bool(y)` , `false` ; otherwise if `bool(x) == false` , `x.get_unexpected() == y.get_unexpected()` ; otherwise `*x == *y` .

Remarks: Specializations of this function template, for which `*x == *y` and `x.get_unexpected() == y.get_unexpected()` 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` and `unexpected_type<E>` shall meet the requirements of *EqualityComparable*.

Returns: If `bool(x) != bool(y)` , `true` ; otherwise if `bool(x) == false` , `x.get_unexpected() != y.get_unexpected()` ; otherwise `*x != *y` .

Remarks: Specializations of this function template, for which `*x != *y` and `x.get_unexpected() != y.get_unexpected()` 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: `*x < *y` and `x.get_unexpected() < y.get_unexpected()` shall be well-formed and its result shall be convertible to `bool` .

Returns: If `!x && y` , false; otherwise, if `x && &y` , true; otherwise, if `&x && &y` , `x.get_unexpected() < y.get_unexpected()` ; otherwise `*x < *y` .

Remarks: Specializations of this function template, for which `*x < *y` and `x.get_unexpected() < y.get_unexpected()` 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: `*x > *y` and `x.get_unexpected() > y.get_unexpected()` shall be well-formed and its result shall be convertible to `bool` .

Returns: If `x && &y` , true; otherwise, if `!x && y` , false; otherwise, if `&x && &y` , `x.get_unexpected() > y.get_unexpected()` ; otherwise `*x > *y` .

Remarks: Specializations of this function template, for which `*x > *y` and `x.get_unexpected() > y.get_unexpected()` 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: `*x <= *y` and `x.get_unexpected() <= y.get_unexpected()` shall be well-formed and its result shall be convertible to `bool`.

Returns: If `!x && y`, false; otherwise, if `x && &y`, true; otherwise, if `&x && &y`, `x.get_unexpected() <= y.get_unexpected()`; otherwise `*x <= *y`.

Remarks: Specializations of this function template, for which `*x > *y` and `x.get_unexpected() > y.get_unexpected()` 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: `*x >= *y` and `x.get_unexpected() >= y.get_unexpected()` shall be well-formed and its result shall be convertible to `bool`.

Returns: If `x && &y`, true; otherwise, if `!x && y`, false; otherwise, if `&x && &y`, `x.get_unexpected() >= y.get_unexpected()`; otherwise `*x >= *y`.

Remarks: Specializations of this function template, for which `*x > *y` and `x.get_unexpected() > y.get_unexpected()` 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);
```

Returns: 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);
```

Returns: bool(x) ? *x != v : false.

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

Returns: `bool(x) ? *x < v : false`.

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

Returns: `bool(x) ? v < *x : true`.

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

Returns: `bool(x) ? *x <= v : false`.

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

Returns: `bool(x) ? v <= *x : true`.

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

Returns: `bool(x) ? *x > v : true`.

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

Returns: `bool(x) ? v > *x : false`.

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

Returns: `bool(x) ? *x >= v : true`.

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

Returns: `bool(x) ? v >= *x : false` .

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

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

Returns: `bool(x) ? true ? x.get_unexpected() == e` .

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

Returns: `bool(x) ? false ? x.get_unexpected() != e` .

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

Returns: `bool(x) ? true : x.get_unexpected() < e` .

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

Returns: `bool(x) ? false : e < x.get_unexpected()` .

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

Returns: `bool(x) ? true : x.get_unexpected() <= e` .

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

Returns: `bool(x) ? false : e < x.get_unexpected()` .

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

Returns: `bool(x) ? false : x.get_unexpected() > e` .

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

Returns: `bool(x) ? true : e > x.get_unexpected()` .

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

Returns: `bool(x) ? false : x.get_unexpected() >= e` .

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

Returns: `bool(x) ? false : e >= x.get_unexpected()` .

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)` .

X.Z.12 Expected Factories [expected.factories]

```
template <class T>
constexpr expected<typename decay<T>::type> make_expected(T&& v);
```

Returns: `expected<typename decay<T>::type>(std::forward<T>(v))` .

```
template <class T, class E>
constexpr expected<T, decay_t<E>> make_expected_from_error(E&& e);
```

Returns: `expected<T, decay_t<E>>(make_unexpected(e))` ;

```
template <class T, class E, class U>
constexpr expected<T, E> make_expected_from_error(U&& u);
```

Returns: `expected<T, E>(make_unexpected(E{forward<U>(u)}))` ;

```
template <class F>
constexpr typename expected<result_of<F()>::type make_expected_from_call(F funct);
```

Equivalent to:

```
try
    return make_expected(funct());
catch (...)
    return make_unexpected_from_current_exception();
```

X.Z.13 Hash support [expected.hash]

```
template <class T, class E>
struct hash<expected<T,E>>;
```

Requires: The template specializations `hash<T>` and `hash<E>` shall meet the requirements of class template `hash` (Z.X.Y). The template specialization `hash<expected<T,E>>` shall meet the requirements of class template `hash`. For an object `e` of type `expected<T,E>`, if `bool(e)`, `hash<expected<T,E>>()(e)` shall evaluate to a combination of the hashing `true` and `hash<T>()(*e)`; otherwise a combination of hashing `false` and `hash<E>()(e.error())`.

```
template <class E>
struct hash<expected<void, E>>;
```

Requires: The template specialization `hash<E>` shall meet the requirements of class template `hash` (Z.X.Y). The template specialization `hash<expected<void,E>>` shall meet the requirements of class template `hash`. For an object `e` of type `expected<void,E>`, if `bool(e)`, `hash<expected<void,E>>()(e)` shall evaluate to the hashing `true`; otherwise it evaluates to a combination of hashing `false` and `hash<E>()(e.error())`.

X.Z.14 `expected` as a meta-fuction [expected.object.meta]

```
template <class E>
class expected<_t, E>
{
public:
    template <class T>
    using apply = expected<T,E>;
};
```

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++14.

An almost full reference implementation of this proposal can be found at ExpectedImpl. However this implementation requires that both `T` and `E` don't throw while constructing and assigning.

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 KrzemieÃĚâĂđski and every one that has contributed to the wording and the rationale of N3793 N3793.

References

- 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

- N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.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

- P0393R3 Tony Van Eerd. Making Variant Greater Equal.

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

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_condition` into a specific exception. Or in debug mode, they might want to use an `assert` call.

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 value() const&
```

About expected

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

- a `T`, or
- an `ErrorCode` throw using `Exception`, or
- a `exception_ptr`

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`. While this would need only an extra test on the exceptional case, the authors think that it is not worth doing it as all the copy/move/swap operations would be less efficient.

Appendix II - Related types

Variant

`expected<T,E>` can be seen as a specialization of `boost::variant<unexpected_type<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_type<E>, T>` and `expected<T,E>`.

	std::variant<T, unexpected_type<E>>	expected<T,E>
never-empty warranty	no	yes
accepts is_same<T,E>	yes	yes
swap	yes	yes
factories	no	make_expected / make_unexpected
hash	yes	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_condition`).

```
template <class T, class E>
expected<T,E> make_expected(optional<T> v) {
    if (v) return make_expected(*v);
    else make_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_type<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	make_expected / make_unexpected	make_ready_future / make_exceptional_future
hash	yes	yes	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