Document number:	N4015
Date:	2014-05-26
Revises:	None
Project:	JTC1.22.32 Programming Language C++
Reply to:	Vicente J. Botet Escriba <vicente.botet@wanadoo.fr> Pierre Talbot <ptalbot@hyc.io>

A proposal to add a utility class to represent expected monad

Contents

1 Introduction **1**

2 Motivation and Scope **1**

3 Use cases **3**

4 Impacts on the Standard **5**

5 Design rationale **5**

6 Related types **12**

7 Open points **14**

8 Proposed Wording **20**

9 Implementability **37**

10 Acknowledgement **37**

1 Introduction

Class template `expected<E,T>` 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 [2]. The interface and the rational are based on `std::optional` N3793 [5] and Haskell monads. We can consider that `expected<E,T>` is a generalization of `optional<T>` providing in addition a monad interface and some specific functions associated to the unexpected type `E`. It requires no changes to core language, and breaks no existing code.

2 Motivation and Scope

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 appears 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 table 1.

	Exception	**Return code**
Visibility	Not visible without further analysis of the code. However, if an exception is thrown, we can follow the stack trace.	Visible at the first sight by watching the prototype of the called function. However ignoring return code can lead to undefined results and it can be hard to figure out the problem.
Informations	Exceptions can be arbitrarily rich.	Historically a simple integer. Nowadays, the header `<system_error>` provides richer error code.
Clean code	Provides clean code, exceptions can be completely invisible for the caller.	Force you to add, at least, a if statement after each function call.
Non-Intrusive	Proper communication channel.	Monopolization of the return channel.

Table 1: Comparison between two error handling systems.

2.1 Expected class

We can do the same analysis for the `Expected<E, T>` 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<E, T> 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<E, T>` 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<E, T>`:

- Associates errors with computational goals.

- Naturally allows multiple errors inflight.

- Teleportation possible.
  - Across thread boundaries.
  - Across nothrow subsystem boundaries.
  - Across time: save now, throw later.

- Collect, group, combine errors.

3 Use cases

3.1 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<E,T>`, 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<error_condition, double>`, the code becomes:

```
expected<error_condition, double> 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<E,T>` and (2) from `T` to `expected<E,T>` 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<error_condition, double>`:

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

This example clearly doesn't respect the "clean code" characteristic introduced in section 2 and the readability doesn't differ much from the "C return code". Hopefully, we can see `expected<E,T>` through functional glasses as a monad. The code is cleaner using the member function `map`. This way, the error handling is not explicitly mentioned but we still know, thanks to the call to `map`, that something is going underneath and thus it is not as silent as exception.

```
expected<error_condition, double> f1(double i, double j, double k)
{
  return safe_divide(j, k).map([&](double q){
    return i + q;
  });
}
```

The `map` member calls the continuation provided if expected contains a value, otherwise it forwards the error to the callee. Using lambda function might clutter the code, so here the same example using functor:

```
expected<error_condition, double> f1(double i, double j, double k)
{
  return safe_divide(j, k).map(bind(plus, i, _1));
}
```

We can use `expected<E, T>` 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<error_condition, int> 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;
}
```

Now we have a division function for integers that possibly fail in two ways. We continue with the exception-oriented function $i/k + j/k$:

```
int f2(int i, int j, int k)
{
  return safe_divide(i,k) + safe_divide(j,k);
}
```

Now let's write this code using an `expected<E,T>` type and the functional `map` already used previously.

```
expected<error_condition, int> f(int i, int j, int k)
{
  return safe_divide(i, k).map([=](int q1) {
      return safe_divide(j,k).map([=](int q2) {
        return q1+q2;
      });
    });
}
```

The compiler will gently say he can convert an `expected<error_condition, expected<error_condition, int>>` to `expected<error_condition, int>`. This is because the member `map` wraps the result in Expected and since we use twice the `map` member it wraps it twice. The `bind`[1] member wraps the result of the continuation only if it is not already wrapped. The correct version is as follow:

```
expected<error_condition, int> f(int i, int j, int k)
{
  return safe_divide(i, k).bind([=](int q1) {
      return safe_divide(j,k).bind([=](int q2) {
        return q1+q2;
      });
    });
}
```

The error-handling code has completely disappeared but the lambda functions are a new source of noise, and this is even more important with n expected variables. Propositions for a better monadic experience are discussed in section 7.1, the subject is left open and is considered out of scope of this proposal.

3.2 Error retrieval and correction

The major advantage of `expected<E,T>` 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`:

[1] To not confound with `std::bind` which is not related to monad.

```
expected<error_condition, int> 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;
}
```

This imperative way is not entirely satisfactory since it suffers from the same disadvantages than `value()`. Again, a functional view leads to a better solution. The `catch_error` member calls the continuation passed as argument if the expected is erroneous.

```
expected<error_condition, int> divide3(int i, int j)
{
  auto e = safe_divide(i,j);
  return e.catch_error([](const error_condition& e){
    if(e.value() == arithmetic_errc::divide_by_zero)
    {
      return 0;
    }
    return make_unexpected(e);
  });
}
```

An advantage of this version is to be coherent with the `bind` and `map` functions. It also provides a more uniform way to analyze error and recover from some of these. Finally, it encourages the user to code its own "error-resolver" function and leads to a code with distinct treatment layers.

4 Impacts on the Standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++ 14. It requires however the `in_place_t` from N3793.

5 Design rationale

The same rationale described in [4] for `optional<T>` applies to `expected<E,T>` and `expected<nullopt_t, T>` should behave as `optional<T>`. That is, we see `expected<E,T>` 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 [4] applied to `expected<E,T>`.

5.1 Conceptual model of `expected<E,T>`

`expected<E,T>` models a discriminated union of types `T` and `unexpected_type<E>`. `expected<E,T>` 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<E,T>` 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` can not be a `nullopt_t`. As the types `T` and `E` could be the same in `expected<E,T>`, 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` can not be `unexpected_type<E>` for a given `E`.

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

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

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

5.2 Initialization of `expected<E,T>`

an In cases `T` and `E` are value semantic types capable of storing `n` and `m` distinct values respectively, `expected<E,T>` can be seen as an extended `T` capable of storing `n + m` values: these that `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 are not `CopyConstructible` and their expected variants still should be constructible with any set of arguments that work for `T`.
As in [4], the model retained is to initialize either by providing an already constructed `T` or a tagged `E`. The default constructor required `E` to be default-constructible (which is more likely to happen than `T`).

```
string s{"STR"};

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

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

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

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

As in [4], and in order to avoid calling move/copy constructor of `T`, we use a “tagged” placement constructor:

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

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

```
expected<string,int> ei{unexpect};           // unexpected value, calls string in place
expected<string,int> 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 not compatible with the default construction of `expected<E,T>` to `E()`, 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`.

5.3 Almost never-empty guaranty

As `boost::variant<unexpected_type<E>,T>`, `expected<E,T>` ensures that it is never empty. All instances `v` of type `expected<E,T>` guarantee that `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 and the significant additional complexity-of-use attendant with such a possibility.

5.3.1 The default constructor

Similar data structure includes `optional<T>`, `variant<`T_1`,...,`T_n`>` and `future<T>`. We can compare how they are default constructed.

- `std::experimental::optional<T>` default constructs to an optional with no value.
- `boost::variant<`T_1`,...,`T_n`>` default constructs to the first type default constructible or it is ill-formed if none are default constructible.

- `std::future<T>` default constructs to an invalid future with no shared state associated, that is, no value and no exception.
- `std::experimental::optional<T>` default constructor is equivalent to `boost::variant<nullopt_t, T>`.

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

- Should the default constructor of `expected<E,T>` behave like `variant<T,E>` or as `variant<E,T>`?
- Should the default constructor of `expected<E,T>` behave like `optional<variant<T,E>>`?
- Should the default constructor of `expected<nullopt_t,T>` behave like `optional<T>`? If yes, how should behave the default constructor of `expected<E,T>`? As if initialized with `make_unexpected(E())`? This would be equivalent to the initialization of `variant<E,T>`.
- Should `expected<E,T>` provide a default constructor at all? [3] presents valid arguments against this approach, e.g. `array<expected<E,T>>` 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 [3]). With the same semantics `expected<E,Date>` would be `Regular` with a meaningful not-a-date state created by default.

There is still a minor issue as the default constructor of `std::exception_ptr` doesn't contains an exception and so getting the value of a default constructed `expected<exception_ptr, T>` would need to check if the stored `std::exception_ptr` is equal to `std::exception_ptr()` and throw a specific exception.

The authors consider the arguments in [3] valid and so propose that `expected<E,T>` default constructor should behave as constructed with `make_unexpected(E())`.

5.3.2 Conversion from `T`

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

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

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

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

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

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

5.3.3 Conversion from `E`

An object of type `E` is not convertible to an unexpected object of type `expected<E,T>` 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<E,T>`. 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` tagged forwarding constructor:

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

exp1 = make_unexpected(1);
exp2 = {unexpect, 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<int, vector<int>> get1() {
  return {unexpect, 1};
}

expected<int, vector<int>> get2() {
  return make_unexpected(1);
}

expected<int, vector<int>> get3() {
  return expected<int, vector<int>>{unexpect, 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>`. While `make_unexpected(E)` has been chosen because it clearly indicates that we are interested in creating an unexpected `expected<E,T>` (of unspecified type `T`), it could be also used to make a ready future with a specific error, but this is outside the scope of this proposal. Note also that the definition of the result type of `make_unexpected` has an explicitly deleted default constructor. This is in order to enable the reset idiom `exp2 = {}` which would otherwise not work due to the ambiguity when deducing the right-hand side argument.

5.4 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.

5.4.1 Explicit conversion to `bool`

The rational described in [4] for `optional<T>` applies to `expected<E,T>` and so, the following example combines initialization and value-checking in a boolean context.

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

5.4.2 Accessing the contained value

Even if `expected<E,T>` has not been used in practice for a while as Boost.Optional, we consider that following the same interface that `std::experimental::optional<T>` makes the C++ standard library more homogeneous. The rational described in [4] for `optional<T>` applies to `expected<E,T>`.

5.4.3 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 that `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.

5.4.4 Function value

In addition to the indirection operator, we propose the member function value as in [4] 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<error,int> ei = str2int(s);

  try {
    process_int(ei.value());
  }
```

```
  catch(bad_expected_access<error>) {
    cout << "this was not a number.";
  }
}
```

The exception thrown depend on the expected error type. By default it throws `bad_expected_access<E>` (derived from `logic_error`) which will contain the stored error. In the case of `expected<exception_ptr>`, it throws the exception stored in the `exception_ptr`. An approach enabling customization of this behavior is presented in the section 7.2.

`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.

5.4.5 Accessing the contained error

Usually, accessing the contained error is done once we know the expected object has no value. This is why the `error()` function has a narrow contract: it works only if `!(*this)`.

```
expected<errc,int> 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`.

5.4.6 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<E, T>{unexpect, f.error()};
```

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

```
expected<errc, pair<int, int>> 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<E>` to `expected<E,T>` and on the other side there is an explicit conversion from `expected<E,T>` to `unexpected<E>`. A more pleasant function manipulating error is `catch_error(F)` and is explained in the monadic operations in section 5.9.

5.4.7 Function `value_or`

The function member `value_or()` has the same semantics than `optional`[4] since the type of `E` doesn't matter; hence we can consider that `E == nullopt_t` and the optional semantics yields. Using the monadic interface, we can achieve a similar behavior:

```
auto x = getInt();
int x = *(x.catch_error([](auto){return 0;})); // identical to x.value_or(0);
```

5.4.8 Relational operators

The relational operators have the same semantics than `optional`[5]. Of course, the error type can be anything but it'll never participate in ordering, thus we can consider than `E == nullopt_t`; hence the semantics of optional can be exactly mapped. We considered that we can compare `unexpected_type` but it seems that errors are not structure that have a semantic ordering. Therefore it is coherent to use the same semantics than `optional`.

5.5 Modifiers

5.5.1 Reseting the value

Reseting the value of `expected<E,T>` is similar to `optional<T>` but instead of building a disengaged `optional<T>`, we build a erroneous `expected<E,T>`. Hence, the semantics and rational is the same than in [4].

5.5.2 Tag `in_place`

This proposal makes use of the "in-place" tag defined in [5]. 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 struct `in_place`. We need the extra tag to disambiguate certain situations, like calling `expected`'s default constructor and requesting `T`'s default construction:

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

5.5.3 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 struct `unexpect`. We need the extra tag to disambiguate certain situations, notably if `T == E`.

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

5.5.4 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<E,T>` have different requirements and may be disable if `T` or `E` doesn't respect these requirements. For example, `expected<E,T>`'s move constructor requires that `T` and `E` are `MoveConstructible`, `expected<E,T>`'s copy constructor requires that `T` and `E` are `CopyConstructible`, and so on. This is because `expected<E,T>` is a wrapper for `T` or `E`: it should resemble `T` as much as possible. If `T` is `EqualityComparable` then (and only then) we expect `expected<E,T>` to be `EqualityComparable`.

5.5.5 Expected references

This proposal doesn't include expected references as `optional`[5] doesn't include references neither.

5.5.6 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.

5.6 Making `expected` a literal type

In [4], 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.

5.7 Moved from state

We follow the approach taken in `optional`[4]. Moving `expected<E,T>` do not modify the state of the source (valued or erroneous) of `expected` and the move semantics is up to `T` or `E`.

5.8 IO operations

For the same reasons than `optional`[4] we do not add `operator<<` and `operator>>` IO operations.

5.9 Monadic operations

A monadic interface is not optional if we don't want to fall back in the problems of the old "C return code". The example section 3.1 shows how these operations are important to expected. The member function `map` and `bind` find their roots in the category theory if we consider `expected` as a functor and a monad.

5.9.1 Functor `map`

The operation `map` consider expected as a functor and just apply a function on the contained value, if any. The types of the two overloads are presented using a functional notation and the `[]` represent a context in which the value T or U is contained. The current context is `expected` and thus `[T]` is equivalent to `expected<E,T>`.

- `(T -> U) -> [U]`
- `(T -> [U]) -> [[U]]`

Whatever the return type of the continuation, we observe that it is always wrapped into a context. The monadic bind do it differently.

5.9.2 Monadic `bind`

A monad is defined with a type constructor and two operations `return` and `bind`. The type constructor simply build a monad for a specific type, in the C++ jargon it is referred to template instantiation (we build `expected` from a type `Value` and `Error`).

The `return` operation wraps a value of type `T` inside a context `[T]`. In C++ we can consider the constructors as a `return` operation.

Finally, the `bind` operation is similar to `map` but doesn't wrap the value if the function already wraps it up. The functional signature of `bind` can be described as follow:

- `(T -> U) -> [U]`
- `(T -> [U]) -> [U]`

If a do-notation is introduced in C++, as proposed in section 7.1, these operations can become a powerful abstraction, they have been proven very useful in Haskell. For example, a similar interface could be used with `optional`..

5.9.3 `then` operation

The last operation has no direct counterpart in functional language and is inspired from [7] proposing some improvements to `std::future<T>`. The functional signature is as follow:

- `([T] -> U) -> [U]`
- `([T] -> [U]) -> [U]`

It has the same wrapping strategy than `bind`: it doesn't wrap if the continuation already wraps it up.

5.9.4 Exception thrown in the continuation

This behavior is left open in the section 7.2. Currently, the exceptions thrown in the continuations are not caught.

5.9.5 `catch_error` operation

This last member function is used when we want to use or recover from an error. When chaining multiple `bind` or `map` operations we don't know if the operations have succeeded. A common way is thus to add a `catch_error` at the end and act in consequence.

```
getInt().map([](int i){return i * 2;})
        .map(integer_divide_by_2)
        .catch_error(log_error);
```

Here the last operation is simply used to log the error but the `catch_error` also accepts function that try to recover from a previous error.

```
getInt().map([](int i){return i * 2;})
        .map(integer_divide_by_2)
        .catch_error([](auto e) { return 0; });
```

This last example shows we can return a new value from the continuation passed to `catch_error`.

The `catch_error` member doesn't catch exceptions that could be thrown by the continuation. Since we already try to recover from an error it makes little sense to prevent the user to launch an exception.

5.9.6 Function `unwrap`

In some scenarios, you might want to create an `expected` that returns another `expected`, resulting in nested expected. It is possible to write simple code to unwrap the outer `expected` and retrieve the nested `expected` and its result with the current interface as in:

```
template <class T, class E>
expected<E,T> unwrap<expected<E, expected<E,T>> ee) {
  if (ee) return *ee;
  return ee.get_unexpected();
}
template <class T, class E>
expected<E,T> unwrap<expected<E,T>> e) {
  return e;
}
```

We could add such a function to the standard, either as a free function or as a member function. The authors propose to add it as a member function to be in line with [7].

6 Related types

6.1 Variant

`expected<E,T>` can be seen as a specialization of `boost::variant<unexpected<E>,T>` which gives a specific intent to its second 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::experimental::optional<T>`. Even if the standard included a class `variant<T,E>`, the interface provided by `expected<E,T>` is more specific and closer to what the user could expect as the result type of a function. In addition, `expected<E,T>` doesn't intend to be used to define recursive data as `boost::variant<>` does.

The table 2 presents a brief comparison between `boost::variant<unexpected<E>, T>` and `expected<E,T>`.

	boost::variant<unexpected<E>, T>	**expected<E,T>**
never-empty warranty	yes	yes
accepts is_same<T,E>	no	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
continuations	apply_visitor	map/bind/then/catch_error

Table 2: Comparison between variant and expected.

6.2 Optional

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

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

We can convert an `optional<T>` to an `expected<E,T>` 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<E,T> make_expected(optional<T> v) {
  if (v) return make_expected(*v);
  else make_unexpected(E());
}
```

6.3 Promise and Future

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

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

We can also create a `future<T>` from an `expected<exception_ptr,T>`.

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

where `make_unexpected_future` is defined as:

```
template <class T, class E>
constexpr future<T> make_unexpected_future(E e)  {
  promise<T> p;
  future<T> f = p.get_future();
  p.set_exception(e);
  return move(f);
}
```

We can combine them as follows:

```
fut.then([](future<int> f) {
  return make_ready_future(
    make_expected(f).bind([](i){ ... }).catch_error(...));
  });
```

As for the `future<T>` proposal, `expected<E,T>` provides also a way to visit the stored values. `future<T>` provides a `then()` function that accepts a continuation having the `future<T>` as parameter. The synchronous nature of expected makes it easier to use two functions, one to manage with the case expected has a value and one to try to recover otherwise. This is more in line with the monad interface, as any function having a `T` as parameter can be used as parameter of the apply function, no need to have a `expected<E,T>`. This make it easier to reuse functions.

	optional	expected	promise/future
specific null value	yes	no	no
relational operators	yes	yes	no
swap	yes	yes	yes
factories	make_optional / nullopt	make_expected / make_unexpected	make_ready_future / (make_exceptional, see [6])
hash	yes	yes	yes
value_type	yes	yes	no / (yes, see [6]).
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 / (has_value, see [6])
observers	pointer-like / value / value_or	pointer-like / value / error / value_or	get / (get_exception_ptr, see [6])
visitation	no	map / bind / then / catch_error	then / (next/recover see [6])
grouping	n/a	n/a	when_all / when_any

Table 3: Comparison between optional, expected and promise/future.

- `expected<E,T>::then()` is the counterpart of `future<T>.then()`
- `expected<E,T>::unwrap()` is the counterpart of `future<T>.unwrap()`
- `expected<E,T>::operator bool()` is the counterpart of `future<T>.has_value()`

6.4 Comparison between optional, expected and future

The table 3 presents a brief comparison between `optional<T>`, `expected<E,T>` and `promise<T>/future<T>`.

7 Open points

7.1 Better support for monad

In the use-cases section (3.1), we present expected as a better way to handle errors than exception or error code. However the current syntax using lambda and chaining monadic operations (such as `map`) can be tedious to use. We propose different solutions to overcome this problem, since the solutions are more general than the scope of this proposal we discuss them in the open points section.

A first solution that do not require change in the language is the use of variadic monadic operation. For example using a variadic free function `map`, we can write the $i/k + j/k$ function as following:

```
expected<exception_ptr,int> f(int i, int j, int k)
{
  return map(plus,
    safe_divide(i, k),
    safe_divide(j, k));
}
```

This is most readable than the member `map` function and the use of lambda. However it suffers from two major deficiencies:

- **Eager evaluation** All arguments are evaluated even if the first fails.
- **Unordered evaluation** We cannot control the order of evaluation thus it presupposed the function to have no side effects.

Considering these two problems we consider a possible C++ language extension: a do-notation similar to the one in Haskell. As with the variadic `map` function, it is not limited to `expected` but could work with any kind of monad. Next follow the grammar:

```
expression ::= ...
            | do-expression
do-expression ::= do-initialization ':' expression
do-initialization ::= type var '<-' expression
```

The previous function could be rewritten as:

```
expected<error_condition, int> f2(int i, int j, int k)
{
  return (
    auto s1 <- safe_divide(i, k) :
    auto s2 <- safe_divide(j, k) :
    s1 + s2
  );
}
```

This syntax is far easier to read and to understand. A lazy evaluation of statement is possible and the order is well-defined. Nevertheless, it can be considered as a syntactic sugar for `bind`. We give a syntactic transformation following:

```
[[do-expression]] =
  bind(expression, [&](type var) {
    return [[do-expression-or-expression]]
  });
```

The transformed code of the previous function is:

```
expected<error_condition, int> f2(int i, int j, int k)
{
  return bind(safe_divide(i, k) ,[=](auto s1) {
    return bind(safe_divide(j, k),[=](auto s2) {
      return s1 + s2;
    });
  });
}
```

This would give the exact same results as the previous version. However, the function `f2` is much simpler and clearer than `f` because it doesn't have to explicitly handle any of the error cases. When an error case occurs, it is returned as the result of the function, but if not, the correct result of a subexpression is bound to a name (`s1` or `s2`), and that result can be used in later parts of the computation. The code is a lot simpler to write. The more complicated the error-handling function, the more important this will be.

But, the standard doesn't have this DO expression yet. Waiting for a do-statement the user could define some macros and define `f2` as

```
expected<exception_ptr,int> f2(int i, int j, int k)
{
    return DO (
         ( s1, safe_divide(i, k) )
         ( s2, safe_divide(j, k) )
         s1 + s2
    );
}
```

In the case of expected and optional, and similarly to the proposed `await` keyword we could use an `expect` keyword (it returns if the expected is not valued):

```
expected<exception_ptr,int> f2(int i, int j, int k)
{
  EXPECT(s1, safe_divide(i, k));
  EXPECT(s2, safe_divide(j, k));
  return s1 + s2;
}
```

Note that this meaning of EXPECT is not valid for the list monad.

7.2 A Configurable Expected

Expected might be configurable through a trait `expected_traits`. The first variation point is the behavior of `value()` when `expected<E,T>` contains an error. The current strategy throw a `bad_expected_access` exception (or the contained exception if the type is `expected<exception_ptr, T>`) but it might not be satisfactory for every error types. 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.

The other variation point is the behavior triggered when the continuation argument of `bind` or `map` throws an exception. If the exception thrown is `system_error` and the error type is `error_code`, we might want to store the error carried by the exception. Without more discussion, let's show how we could customize `expected<E, T>`, consider the following exception-oriented function:

```
class error_cond : public std::exception {
  // Implementation similar to system_error but for error_condition here.
};

int safe_divide(int i, int j)
{
  if (j == 0)
    throw error_cond(error_condition(arithmetic_errc::divide_by_zero));
  return i/j;
}
```

Imagine `j` encapsulated into an `expected`, you will call `map` with `safe_divide` as the continuation. Let's see what it looks like:

```
expected<error_condition, int> f(int i, const expected<error_condition, int>& j)
{
  return j.map(bind(safe_divide, i, _1));
}
```

If we specialize `expected_traits` for `error_condition`, we can achieve the expected behavior:

```
template <class T>
struct expected_traits<expected<error_condition, T>>
{
  static expected<error_condition, T> catch_exception(exception_ptr e)
  {
    try{
      rethrow_exception(e);
    } catch(const error_cond& e) {
      return make_unexpected(e.code());
    }
  }

  static void bad_access(const error_type &e)
  {
    throw error_cond(e);
  }
};
```

The semantics of `catch_exception` is to rethrow the current exception and catch only the exceptions we are interested in. The default behavior let flight the exception thrown by the continuation. We created a bridge between an `error_condition` and the `error_cond` exception.

7.3 Allocator support

As `optional<T>`, `expected<E,T>` does not allocate memory. So it can do without allocators. However, it can be useful in compound types like:

```
typedef vector<expected<error, vector<int, MyAlloc>>, MyAlloc> MyVec;
MyVec v{ v2, MyAlloc{} };
```

One could expect that the allocator argument is forwarded in this constructor call to the nested vectors that use the same allocator. Allocator support would enable this. `std::tuple` offers this functionality.

7.4 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()`

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

where

```
template <class Exception, class E, class T>
constexpr value_type value_or_throw(expected<E,T>& e) const&
{
  return *this
    ? move(**this)
    : throw Exception(e.error());
}
```

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

The user can also wrap the proposed class in its own expected class

```
template <class T, class Error=std::error_code, class Exception=std::system_error>
struct MyExpected {
  expected <T,E> v;
  MyExpected(expected <T,E> v) : v(v) {}
  T value() {
    if (e) return v.value();
    else throw Exception(v.error());
  }
  ...
};
```

and use it as

```
std::experimental::expected<std::error_code, int> f();
MyExpected<int> e = f();
auto i = e.value();  // std::system_error throw if not valid
```

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

An alternative could be to add a specialization on a error class that gives the storage and the exception to thrown.

```
template <class Error, class Exception>
  struct error_exception {
    typedef Error error_type;
    typedef Exception exception_type;
  };
```

that could be used as follows

```
std::experimental::expected<std::error_exception<std::error_code, std::system_error>, T> e = make_unexpected(err
e.value(); // will throw std::system_error(err);
```

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

7.5 About `expected<T, ErrorCode, Exception>`

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

- a T, or

- an error code, 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.

7.6 Should expected<exception_ptr,T> be equality comparable?

This proposal doesn't make expected<exception_ptr,T> comparable as exception_ptr equality comparison is shallow and doesn't provides relational operators.

Should expected<exception_ptr,T> be equality comparable using shallow comparison?

7.7 Should expected<E,T> make all the unexpected values equal?

Currently expected<E,T> and expected<exception_ptr,T> don't compare its values in the same way. Should expected<E,T> make all the unexpected values equal as it does expected<exception_ptr,T>?

7.8 Should expected<E,T> and expected<exception_ptr,T> be represented by two different classes?

As the behavior of expected<E,T> and expected<exception_ptr,T> differs in a lot of points (at least currently), should expected<E,T> and expected<exception_ptr,T> be represented by two different classes?

7.9 Should expected<E,T> throw E instead of bad_expected_access<E>?

As any type can be thrown as an exception, should expected<E,T> throw E instead of bad_expected_access<E>? If yes, should optional<T> throw nullopt_t to be coherent?

7.10 Should expected<E,T> be convertible from E when E it is not convertible to T?

The implicit conversion from E has been forbidden to avoid ambiguity when E and T are the same type. However when E it is not convertible to T there wouldn't any ambiguity. Should the implicit conversion be allowed in this case?

7.11 Should a specific exception be thrown when the expected<exception_ptr,T> doesn't have a value neither an exception stored?

The following call in (1) is undefined behavior. Should a specific exception be thrown?

```
expected<exception_pre,int> e;
e.value(); // (1)
```

7.12 Should map/bind/then catch the exceptions throw by the continuation?

It is easy to catch the exceptions when the type is expected<exception_ptr,T>. However, doing it for expected<E,T> needs a conversion from the current exception and the error E.

This proposal requires that the continuation doesn't throw exceptions as we don't have a general solution.

Should expected<exception_ptr,T>::map/bind/then catch the exceptions and propagate them on the result?

Should expected<E,T>::map/bind/then catch the exceptions and propagate them on the result by doing a transformation from the exception to the error? If yes, how to configure it?

7.13 Should the order of parameters be reversed expected<T,E> instead of expected<E,T>?

It has been argued that the name of the class expected would need as first parameter the value type and not the error type.

The original class was defined just in this way expected<T,E>. We moved to the current expected<E,T> form to be able to write

```
auto e  = expected<error_condition>::make(1) // result is expected<error_condition, int>
```

We can comeback to the original design. However we need to remove this `expected<E>::make(T)` function.

We could add it on a specific type constructor `expected_tc` (a better name of course)

```
auto e  = expected_tc<error_condition>::make(1) // result would be expected<int, error_condition>.
```

7.14 Do we need a `expected<E,T>::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 <E,T>` would be seen more like something

```
struct { E; optional<T> }.
```

The following code was show as 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.status()) {
  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<E,T> const&, E err) {
  if(e) return err;
  else return error();
}
```

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

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

An other use case which could look much uglier is if the user had to test for whether or not there was a status.

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

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;
}
```

where

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

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

8 Proposed Wording

The proposed changes are expressed as edits to N3908, the Working Draft - C++ Extensions for Library Fundamentals [1]. The wording has been adapted from the section "Optional objects".

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 <experimental/unexpected> synopsis [unexpected.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
  // X.Y.3, Unexpected object type
  template <class E>
  struct unexpected_type;
  // X.Y.4, Unexpected exception_ptr specialization
  template <>
  struct unexpected_type<exception_ptr>;

  // X.Y.5, Unexpected factories
  template <class E>
  constexpr unexpected_type<decay_t<E>> make_unexpected(E&& v);
  unexpected_type<std::exception_ptr> make_unexpected_from_current_exception();
}}}
```

A program that necessitates 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=std::exception_ptr>
class unexpected_type {
public:
    unexpected_type() = delete;
    constexpr explicit unexpected_type(E const&);
    constexpr explicit unexpected_type(E&&);
    constexpr E const& value() const;
private:
    E val; // exposition only
};
```

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

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 E const& value() const;
```

Returns:

`val`.

X.Y.4 Unexpected exception_ptr specialization [unexpected.exception_ptr]

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

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

Effects:

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

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

Effects:

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

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

Effects:

Build an unexpected storing the result of `val(make_exception_ptr(e))`.

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

Returns:

`val`.

X.Y.5 Factories [unexpected.factories]

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

Returns:

`unexpected<decay_t<E>>(v)`.

```
constexpr unexpected_type<std::exception_ptr> make_unexpected_from_current_exception();
```

Returns:

`unexpected<std::exception_ptr>(std::current_exception())`.

Insert a new section.

X.Y Expected objects [expected]

X.Y.6 In general [expected.general]

This subclause describes class template expected that represents expected objects. An `expected<E,T>` 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.Y.7 Header <experimental/expected> synopsis [expected.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
  // ??, holder class used as default.
  class holder;
  // X.Y.9, expected for object types
  template <class E= exception_ptr, class T=holder>
  class expected;
  // X.Y.11, Specialization for void.
  template <class E>
  class expected<E, void>;
  // X.Y.10, Specialization of expected as a meta-function : T-> expected<E,T>.
  template <class E>
  class expected<E, holder>;

  // X.Y.12, unexpect tag
  struct unexpect_t{};
  constexpr unexpet_t unexpect{};

  // X.Y.13, class bad_expected_access
  class bad_expected_access;


  // X.Y.14, Expected relational operators
  template <class T, class E>
    constexpr bool operator==(const expected<E,T>&, const expected<E,T>&);
  template <class T, class E>
    constexpr bool operator!=(const expected<E,T>&, const expected<E,T>&);
  template <class T, class E>
    constexpr bool operator<(const expected<E,T>&, const expected<E,T>&);
  template <class T, class E>
    constexpr bool operator>(const expected<E,T>&, const expected<E,T>&);
  template <class T, class E>
    constexpr bool operator<=(const expected<E,T>&, const expected<E,T>&);
  template <class T, class E>
    constexpr bool operator>=(const expected<E,T>&, const expected<E,T>&);

  // X.Y.15, Comparison with T
  template <class T, class E> constexpr bool operator==(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator==(const T&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator!=(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator!=(const T&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator<(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator<(const T&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator<=(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator<=(const T&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator>(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator>(const T&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator>=(const expected<E,T>&, const T&);
  template <class T, class E> constexpr bool operator>=(const T&, const expected<E,T>&);

  // X.Y.16, Comparison with unexpected_type<E>
  template <class T, class E> constexpr bool operator==(const expected<E,T>&, const unexpected<E>&);
  template <class T, class E> constexpr bool operator==(const unexpected<E>&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator!=(const expected<E,T>&, const unexpected<E>&);
  template <class T, class E> constexpr bool operator!=(const unexpected<E>&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator<(const expected<E,T>&, const unexpected<E>&);
  template <class T, class E> constexpr bool operator<(const unexpected<E>&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator<=(const expected<E,T>&, const unexpected<E>&);
  template <class T, class E> constexpr bool operator<=(const unexpected<E>&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator>(const expected<E,T>&, const unexpected<E>&);
```

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


  // X.Y.17, Specialized algorithms
  template <class T>
    void swap(expected<E,T>&, expected<E,T>&) noexcept(see below);

  // X.Y.18, Factories
  template <class T> constexpr expected<exception_ptr, decay_t<T>> make_expected(T&& v);
  template <> expected<exception_ptr, void> make_expected();
  template <class E> expected<E,void> make_expected();

  template <class T>
  expected_type<T> make_expected_from_current_exception();
  template <class T, class E>
   constexpr expected<exception_ptr,T> make_expected_from_exception(E e);
  template <class T>
   constexpr expected<exception_ptr,T> make_expected_from_exception(std::exception_ptr v);

  template <class T, class E>
   constexpr expected<decay_t<E>,T> make_expected_from_error(E v);

  template <class F>
   constexpr typename expected<exception_ptr, typename result_type<F>::type
   make_expected_from_call(F f);

  // X.Y.19, hash support
  template <class T> struct hash;
  template <class T> struct hash<expected<E,T>>;
}}}
```

A program that necessitates the instantiation of template `expected<E,T>` 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.Y.8 Definitions [expected.defs]

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

X.Y.9 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 {
      typedef expected<error_type, U> type;
    };

     // X.Y.9.1, 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&);

// X.Y.9.2, destructor
~expected();

// X.Y.9.3, 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.Y.9.4, swap
void swap(expected&) noexcept(see below);

// X.Y.9.5, observers
constexpr T const* operator ->() const;
T* operator ->();

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

constexpr explicit operator bool() const noexcept;

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

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

constexpr unexpected<E>  get_unexpected() const;

template <typename Ex>
bool has_exception() const;

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

template constexpr 'see below' unwrap() const&;
template 'see below' unwrap() &&;

// X.Y.9.6, factories

template <typename Ex, typename F>
expected<E,T> catch_exception(F&& f);

template <typename F>
  expected<E, decltype(func(declval<T>()))> map(F&& func) ;
template <typename F>
  'see below' bind(F&& func);
template <typename F>
  expected<E,T> catch_error(F&& f);
template <typename F>
```

```
    'see below' then(F&& func);

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

Valued instances of `expected<E,T>` 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<E,T>` 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.Y.9.1 Constructors [expected.object.ctor]

`constexpr expected<E,T>::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<E,T>::expected(const expected<E,T>& 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<E,T>::expected(expected<E,T> && 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 == trye` 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<E,T>::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<E,T>::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<E,T>::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:

`! *this`.

Throws:

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

Remarks:

If `E`'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<E>::value`.

```
constexpr expected<E,T>::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:

`! *this`.

Throws:

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

Remarks:

If `E`'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<E>::value`.

X.Y.9.2 Destructor [expected.object.dtor]

`expected<E,T>::~expected();`

Effects:

If `is_trivially_destructible<T>::value != true` and `bool(*this)`, calls `val->T::~T()`.
If `is_trivially_destructible<E>::value != true` and `! *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.Y.9.3 Assignment [expected.object.assign]

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

Effects:

if `bool(*this)` and `bool(rhs)`, assigns `*rhs` to the contained value `val`, otherwise
if `bool(*this)` and `! rhs`, destroys the contained value by calling `val->T::~T()` and initializes the contained value as if direct-non-list-initializing an object of type `E` with `rhs.error()`, otherwise
if `! *this` and `! rhs`, assigns `rhs.error()` to the contained value `err`, otherwise
if `! *this` and `bool(rhs)`, destroys the contained value by calling `err->E::~E()` and initializes the contained value as if direct-non-list-initializing an object of type `E` with `rhs.error()`.

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. If an exception is thrown during the call to `E`'s copy constructor, no effect. If an exception is thrown during the call to `E`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `E`'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<E,T>& expected<E,T>::operator=(expected<E,T>&& rhs) noexcept(`*/*see below*/*`);`

Effects:

if `bool(*this)` and `rhs` is values, assigns `std::move(*rhs)` to the contained value `val`, otherwise
if `bool(*this)` and `! rhs`, destroys the contained value by calling `val->T::~T()` and initializes the contained value as if direct-non-list-initializing an object of type `E` with `rhs.error()`, otherwise
if `! *this` and `! rhs`, assigns `std::move(rhs.error())` to the contained value `err`, otherwise if `! *this` and `bool(rhs)`, destroys the contained value by calling `err->E::~E()` and initializes the contained value as if direct-non-list-initializing an object of type `E` with `rhs.error()`.

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_move_constructible<E>::value`.

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 move constructor, the state of `rhs.val` is determined by exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `val` and `rhs.val` is determined by exception safety guarantee of `T`'s move assignment. If an exception is thrown during the call to `E`'s move constructor, the state of `rhs.err` is determined by exception safety guarantee of `E`'s move constructor. If an exception is thrown during the call to `E`'s move assignment, the state of `err` and `rhs.err` is determined by exception safety guarantee of `E`'s move 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<E,T>& expected<E,T>::operator=(U&& v);
```

Effects:

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

Returns:

`*this`.

Postconditions:

`bool(*this)`.

Exception Safety:

If any exception is thrown, `bool(*this)` remains unchanged. If an exception is thrown during the call to `E`'s constructor, the state of `e` is determined by exception safety guarantee of `E`'s constructor. If an exception is thrown during the call to `E`'s assignment, the state of `err` and `e` is determined by exception safety guarantee of `E`'s assignment.

Remarks:

This signature shall not participate in overload resolution unless
`is_constructible<T,U>::value` and
`is_assignable<T&, U>::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<E,T>& expected<E,T>::operator=(unexpected_type<E>&& e);
```

Effects:

If `! *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:

`! *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 expected<E,T>::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 constructor, `*this` is disengaged, and the previous `val` (if any) has been destroyed.

Throws:

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

Remarks:

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

```
template <class U, class... Args>
void expected<E,T>::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 constructor, `! *this` , and the previous `val` (if any) has been destroyed.

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`.

X.Y.9.4 Swap [expected.object.swap]

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

Effects:

if `bool(*this)` and `bool(rhs)`, calls `swap(val, rhs.val)`, otherwise
if `! *this` and `! rhs`, calls `swap(err, rhs.err)`, otherwise
if `bool(*this)` and `! 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:

TODO: This must be worded.

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 && noexcept(swap(declval<T&>(), declval<T&>())) &&`
`is_nothrow_move_constructible<E>::value && noexcept(swap(declval<E&>(), declval<E&>()))`.

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.Y.9.5 Observers [expected.object.observe]

```
constexpr T const* expected<E,T>::operator->() const;
T* expected<E,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 T const& expected<E,T>::operator *() const&;
T& expected<E,T>::operator *() &;
```

Requires:

`bool(*this)`.

Returns:

`val`.

Remarks:

The first function shall be a `constexpr` function.

```
constexpr T&& expected<E,T>::operator *() &&;
```

Requires:

`bool(*this)`.

Returns:

`move(val)`.

Remarks:

This function shall be a `constexpr` function.

```
constexpr explicit expected<E,T>::operator bool() noexcept;
```

Returns:

`has_value`.

Remarks:

This function shall be a `constexpr` function.

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

Returns:

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

Throws:

- When `E` is `std::exception_ptr` as if `rethrow_exception(error())` if `!*this`
- Otherwise `bad_expected_access(err)` if `!*this`.

Remarks:

The first and third functions shall be `constexpr` functions.

```
constexpr E const& expected<E,T>::error() const&;
constexpr E& expected<E,T>::error() &;
```

Requires:

`!*this`.

Returns:

`err`.

Remarks:

The first function shall be a `constexpr` function.

```
constexpr E&& expected<E,T>::error() &&;
```

Requires:

`!*this`.

Returns:

`move(err)`.

Remarks:

The first function shall be a `constexpr` function.

```
template <class Ex>
bool expected<E,T>::has_exception() const;
```

Returns:

`true` if and only if `!(*this)` and the stored exception is a base type of `Ex`.

```
constexpr unexpected<E> expected<E,T>::get_unexpected() const;
```

Requires:

`!*this`.

Returns:

`make_unexpected(err)`.

```
template <class U>
constexpr T expected<E,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 expected<E,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`.

```
  template <class E, class U>
  constexpr expected<E,U> expected<E,expected<E,U>>::unwrap() const&;
>::unwrap() &&;
```

Returns:

If `bool(*this)` then `**this`. else `get_unexpected()`

Throws:

Any exception thrown by the selected constructor of `expected<E,U>`.

Remarks:

The function shall not participate in overload resolution unless: `is_copy_constructible<expected<E,T>>::value`

```
  template <class E, class T>
  constexpr expected<E,T> expected<E,T>::unwrap() const&;
>::unwrap() &&;
```

Returns:

`*this`.

Throws:

Any exception thrown by the selected constructor of `expected<E,T>`.

Remarks:

The function shall not participate in overload resolution unless: `T` is not `expected<E,U>` and `is_copy_constructible<expected<E,T>>::value`

```
  template <class E, class U>
  expected<E,T> expected<E,  expected<E,U>>::unwrap() &&;
>::unwrap() &&;
```

Returns:

If `bool(*this)` then `std::move(**this)`. else `get_unexpected()`

Throws:

Any exception thrown by the selected constructor of `expected<E,U>`.

Remarks:

The function shall not participate in overload resolution unless: `is_move_constructible<expected<E,U>>::value`

```
  template <class E, class T>
  template expected<E,T> expected<E,T>::unwrap() &&;
>::unwrap() &&;
```

Returns:

`std::move(**this)`.

Throws:

Any exception thrown by the selected constructor of `expected<E,T>`.

Remarks:

The function shall not participate in overload resolution unless: `is_move_constructible<expected<E,T>>::value`

X.Y.9.6 Factories [expected.object.factories]

```
template <class Ex,class F>
expected<E,T> expected<E,T>::catch_exception(F&& func);
```

Effects:

if `has_exception<Ex>()` call the continuation function `fuct` with the stored exception as parameter.

Returns:

if `has_exception<Ex>()` returns the result of the call continuation function `fuct` possibly wrapped on a `expected<E,T>`, otherwise, returns `*this`.

```
template <class Ex,class F>
'see below' expected<E,T>::map(F&& func)
```

Returns:

if `bool(*this)` returns returns `expected<E, decltype(func(move(val)))>(func(move(val)))`, otherwise, returns `get_unexpected()`.

```
template <class Ex,class F>
'see below' expected<E,T>::bind(F&& func)
```

Returns:

if `bool(*this)` returns returns `unwrap(expected<E, decltype(func(move(val)))>(func(move(val))))`, otherwise, returns `get_unexpected()`.

```
template <class Ex,class F>
'see below' expected<E,T>::then(F&& func);
```

Returns:

returns `unwrap(expected<E, decltype(func(move(*this)))>(func(move(*this))))`,

```
template <class Ex,class F>
expected<E,T> expected<E,T>::catch_error(F&& func);
```

Returns:

if `! (*this)` returns `unwrap(expected<E, decltype(func(val))>(funct(**this)))`, if `! *this` returns the result of the call continuation function `fuct` possibly wrapped on a `expected<E,T>`, otherwise, returns `*this`.

X.Y.10 `expected` as a meta-fuction [expected.object.meta]

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

X.Y.11 expected for void [expected.object.void]

```
template <class E>
class expected<E, void>
{
public:
  typedef void value_type;
  typedef E error_type;

  template <class U>
  struct rebind {
    typedef expected<error_type, U> 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;
    void value() const;
    constexpr E const& error() const&;
    constexpr E& error() &;
    constexpr E&& error() &&;
    constexpr unexpected<E>  get_unexpected() const;

    template <typename Ex>
    bool has_exception() const;

    template constexpr 'see below' unwrap() const&;
    template 'see below' unwrap() &&;

    // ??, factories


    template <typename Ex, typename F>
    expected<E,void> catch_exception(F&& f);

    template <typename F>
      expected<E, decltype(func())> map(F&& func) ;
    template <typename F>
      'see below' bind(F&& func) ;
    template <typename F>
      expected<void,E> catch_error(F&& f);
    template <typename F>
      'see below' then(F&& func);

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

TODO: Describe the functions.

X.Y.12 unexpect tag [expected.unexpect]

```
struct unexpet_t{};
constexpr unexpet_t unexpet{};
```

X.Y.13 Template Class bad_expected_access [expected.bad_expected_access]

```
namespace std {
  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 E const& bad_expected_access::error() const;
E& bad_expected_access::error();
```

Returns:
The stored error..

Remarks:
The first function shall be a `constexpr` function.

X.Y.14 Expected Relational operators [expected.relational_op]

TODO: Describe the functions.

X.Y.15 Comparison with `T` [expected.comparison_T]

TODO: Describe the functions.

X.Y.16 Comparison with `unexpected<E>` [expected.comparison_unexpected_E]

TODO: Describe the functions.

X.Y.17 Specialized algorithms [expected.specalg]

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

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

X.Y.18 Expected Factories [expected.factories]

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

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

```
expected<exception_ptr, void> make_expected();
template <class E>
expected<E, void> make_expected();
```

Returns:
`expected<E,void>(in_place)`.

```
template <class T>
expected<exception_ptr,T> make_expected_from_exception(std::exception_ptr v);
```

Returns:

`expected<exception_ptr,T>(unexpected_type<E>(std::forward<E>(v)))`.

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

Returns:

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

```
template <class T>
constexpr expected<exception_ptr,T> make_expected_from_current_exception();
```

Returns:

`expected<exception_ptr,T>(make_unexpected_from_current_exception())`

```
template <class F>
constexpr typename expected<exception_ptr, 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.Y.19 Hash support [expected.hash]

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

Requires:

The template specialization `hash<T>` and `hash<E>` shall meet the requirements of class template `hash` (Z.X.Y). The template specialization `hash<expected<E,T>>` shall meet the requirements of class template `hash`. For an object `e` of type `expected<E,T>`, `if bool(e)`, `hash<expected<E,T>>()(e)` shall evaluate to the same value as `hash<T, E>()(*e)`; otherwise it evaluates to an unspecified value if `E` is `exception_ptr` or `hash<T, E>()(e.error());`.

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

Requires:

9 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 TBoost.Expected [8].

10 Acknowledgement

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 [5].

Vicente thanks personnaly Evgeny Panasyuk and Johannes Kapfhammer for their remarks on the DO-expression.

Pierre thanks the IRCAM and Carlos Agon who allowed him to work on this proposal during its internship.

References

[1] N3908 - working draft - c++ extensions for library fundamentals, March 2014. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3908.html#optional.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Vicente J. Botet Escriba. N3865, more improvements to std::future<t>, 2013. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3865.pdf.

[7] H. Sutter S. Mithani N. Gustafsson, A. Laksberg. N3857, improvements to std::future<t> and related apis, 2013. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3857.pdf.

[8] Pierre Talbot and Vicente J. Botet Escriba. TBoost.Expected, 2014. https://github.com/ptal/Boost.Expected.