A proposal to add a utility class to represent expected monad
Revision 1

Contents

1 History 1
2 Introduction 1
3 Motivation and Scope 2
4 Use cases 3
5 Impacts on the Standard 5
6 Design rationale 5
7 Related types 14
8 Open points 16
9 Proposed Wording 23
10 Implementability 41
11 Acknowledgement 41

1 History

- R1- Revision of N4015 [9] after Rapperswil feedback:
  - Switch the expected class template parameter order from expected<E,T> to expected<T,E>.
  - Make the unexpected value a salient attribute of the expected class concerning the relational operators.
  - Removed open point about making expected<T,E> and expected<T> different classes.

2 Introduction

Class template expected<T,E> proposed here is a type that may contain a value of type T or a value of type E in its storage space. T represents the expected value, E represents the reason explaining why it doesn’t contains a value of type T, that is the unexpected value. Its interface allows to query if the underlying value is either the expected value (of type T) or an unexpected value (of type E). The original idea comes from Andrei Alexandrescu C++ and Beyond 2012: Systematic Error Handling in C++ talk [2]. The interface and the rational are based on std::optional N3793 [5] and Haskell monads. We can consider that expected<T,E> 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.
3 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 appear throughout the code review. Because the debug can be painful if the errors are hidden.
- **Information on errors** The errors should carry out as much 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.

<table>
<thead>
<tr>
<th></th>
<th>Exception</th>
<th>Return code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visibility</strong></td>
<td>Not visible without further analysis of the code. However, if an exception is thrown, we can follow the stack trace.</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Informations</strong></td>
<td>Exceptions can be arbitrarily rich.</td>
<td>Historically a simple integer. Nowadays, the header <code>&lt;system_error&gt;</code> provides richer error code.</td>
</tr>
<tr>
<td><strong>Clean code</strong></td>
<td>Provides clean code, exceptions can be completely invisible for the caller.</td>
<td>Force you to add, at least, a if statement after each function call.</td>
</tr>
<tr>
<td><strong>Non-Intrusive</strong></td>
<td>Proper communication channel.</td>
<td>Monopolization of the return channel.</td>
</tr>
</tbody>
</table>

Table 1: Comparison between two error handling systems.

3.1 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.

4 Use cases

4.1 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:

```cpp
struct DivideByZero: public std::exception {...};

double safe_divide(double i, double j)
{
  if (j==0) throw DivideByZero();
  else return i / j;
}
```

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

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

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

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

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

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

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

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

This example clearly doesn’t respect the “clean code” characteristic introduced in section 3 and the readability doesn’t differ much from the “C return code”. Hopefully, we can see `expected<T,E>` 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<double, error_condition> 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<double, error_condition> f1(double i, double j, double k)
{
    return safe_divide(j, k).map(bind(plus, i, _1));
}

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

expected<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<T,E> type and the functional map already used previously.

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

The compiler will gently say he can convert an expected<expected<int, error_condition>, error_condition> to expected<int, error_condition>. This is because the member map wraps the result in Expected and since we use twice the map member it wraps it twice. The bind1 member wraps the result of the continuation only if it is not already wrapped. The correct version is as follow:

expected<int, error_condition> 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 8.1, the subject is left open and is considered out of scope of this proposal.

1To not confound with std::bind which is not related to monad.
4.2 Error retrieval and correction

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

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

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

We spoke about how to use the value contained in the Expected but didn’t discuss yet the error usage. A first imperative way to use our error is to simply extract it from the Expected using the error() member function. The following example shows a divide2 function that return 0 if the error is divide_by_zero:

```cpp
expected<int, error_condition> divide2(int i, int j)
{
    auto e = safe_divide(i,j);
    if (!e && e.error().value() == arithmetic_errc::divide_by_zero) {
        return 0;
    }
    return e;
}
```

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.

```cpp
expected<int, error_condition> 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.

5 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.

6 Design rationale

The same rationale described in [4] for optional<T> applies to expected<T,E> and expected<T, nullopt_t> should behave as optional<T>. That is, we see expected<T,E> as optional<T> for which all the values of E collapse into a single value nullopt. In the following sections we present the specificities of the rationale in [4] applied to expected<T,E>. 

5
6.1 Conceptual model of \texttt{expected<T,E>}

\texttt{expected<T,E>} models a discriminated union of types \texttt{T} and \texttt{unexpected_type<E>}. \texttt{expected<T,E>} is viewed as a value of type \texttt{T} or value of type \texttt{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 \texttt{T}, comparison to \texttt{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 (\texttt{T} or \texttt{E}) and either the value of \texttt{T} or the value of \texttt{E}.

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

\begin{verbatim}
expected<int, string> ei = 0;
expected<int, string> ej = 1;
expected<int, string> ek = make_unexpected(string());

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

ei = make_unexpected(E());
ej = 0;
ek = 1;
\end{verbatim}

6.2 Initialization of \texttt{expected<T,E>}

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

As in [4], the model retained is to initialize either by providing an already constructed \texttt{T} or a tagged \texttt{E}. The default constructor required \texttt{E} to be default-constructible (which is more likely to happen than \texttt{T}).

\begin{verbatim}
string s"STR";

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

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

In order to create an unexpected object, the special function \texttt{make_unexpected} needs to be used:

\begin{verbatim}
expected<string, int> epmake_unexpected(-1); // unexpected value, requires Movable<E>
expected<string, int> eq = make_unexpected(-1); // unexpected value, requires Movable<E>
\end{verbatim}

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

\begin{verbatim}
expected<MoveOnly, error_condition> eg;       // unexpected value
expected<MoveOnly, error_condition> eh{};     // unexpected value
expected<MoveOnly, error_condition> eiln_place; // calls MoveOnly{} in place
expected<MoveOnly, error_condition> ej{in_place, "arg"}; // calls MoveOnly("arg") in place
\end{verbatim}

To avoid calling move/copy constructor of \texttt{E}, we use a “tagged” placement constructor:

\begin{verbatim}
expected<int, string> ei(unexpect);        // unexpected value, calls string{} in place
expected<int, string> ej(unexpect, "arg"); // unexpected value, calls string("arg") in place
\end{verbatim}

\texttt{optional<T>}, T can not be a \texttt{nullopt_t}. As the types \texttt{T} and \texttt{E} could be the same in \texttt{expected<T,E>}, there is need to tag the values of \texttt{E} to avoid ambiguous expressions. The \texttt{make_unexpected(E)} function is proposed for this purpose. However \texttt{T} can not be \texttt{unexpected_type<E>} for a given \texttt{E}.

\begin{verbatim}
expected<int, string> ei = 0;
expected<int, string> ej = 1;
expected<int, string> ek = make_unexpected(string());

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

ei = make_unexpected(E());
ej = 0;
ek = 1;
\end{verbatim}

6.2 Initialization of \texttt{expected<T,E>}

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

As in [4], the model retained is to initialize either by providing an already constructed \texttt{T} or a tagged \texttt{E}. The default constructor required \texttt{E} to be default-constructible (which is more likely to happen than \texttt{T}).

\begin{verbatim}
string s"STR";

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

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

In order to create an unexpected object, the special function \texttt{make_unexpected} needs to be used:

\begin{verbatim}
expected<string, int> epmake_unexpected(-1); // unexpected value, requires Movable<E>
expected<string, int> eq = make_unexpected(-1); // unexpected value, requires Movable<E>
\end{verbatim}

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

\begin{verbatim}
expected<MoveOnly, error_condition> eg;       // unexpected value
expected<MoveOnly, error_condition> eh{};     // unexpected value
expected<MoveOnly, error_condition> eiln_place; // calls MoveOnly{} in place
expected<MoveOnly, error_condition> ej{in_place, "arg"}; // calls MoveOnly("arg") in place
\end{verbatim}

To avoid calling move/copy constructor of \texttt{E}, we use a “tagged” placement constructor:

\begin{verbatim}
expected<int, string> ei(unexpect);        // unexpected value, calls string{} in place
expected<int, string> ej(unexpect, "arg"); // unexpected value, calls string("arg") in place
\end{verbatim}
An alternative name for \texttt{in\_place} that is coherent with \texttt{unexpect} could be \texttt{expect}. Being compatible with \texttt{optional<T>} seems more important. So this proposal doesn’t propose such a \texttt{expect} tag.

The alternative and also comprehensive initialization approach, which is not compatible with the default construction of \texttt{expected<T,E>} to \texttt{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 \texttt{T}.

6.3 Almost never-empty guaranty

As \texttt{boost::variant<unexpected\_type<E>,T>, expected<T,E>} ensures that it is never empty. All instances \(v\) of type \texttt{expected<T,E>} 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 \texttt{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.

6.3.1 The default constructor

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

- \texttt{std::experimental::optional<T>} default constructs to an optional with no value.
- \texttt{boost::variant<T1,...,Tn>} default constructs to the first type default constructible or it is ill-formed if none are default constructible.
- \texttt{std::future<T>} default constructs to an invalid future with no shared state associated, that is, no value and no exception.
- \texttt{std::experimental::optional<T>} default constructor is equivalent to \texttt{boost::variant<nullopt_t,T>}.

It raises several questions about \texttt{expected<T,E>}:

- Should the default constructor of \texttt{expected<T,E>} behave like \texttt{variant<T,E>} or as \texttt{variant<E,T>}?
- Should the default constructor of \texttt{expected<T,E>} behave like \texttt{optional<variant<T,E>?>}?
- Should the default constructor of \texttt{expected<nullopt_t,T>} behave like \texttt{optional<T>}? If yes, how should behave the default constructor of \texttt{expected<T,E>}? As if initialized with \texttt{make_unexpected(E())}? This would be equivalent to the initialization of \texttt{variant<E,T>}.
- Should \texttt{expected<T,E>} provide a default constructor at all? [3] presents valid arguments against this approach, e.g. \texttt{array<expected<T,E>>} would not be possible.

Requiring \(E\) to be default constructible seems less constraining than requiring \(T\) to be default constructible (e.g. consider the \texttt{Date} example in [3]). With the same semantics \texttt{expected<Date,E>} would be \texttt{Regular} with a meaningful not-a-date state created by default.

There is still a minor issue as the default constructor of \texttt{std::exception\_ptr} doesn’t contains an exception and so getting the value of a default constructed \texttt{expected<T>} would need to check if the stored \texttt{std::exception\_ptr} is equal to \texttt{std::exception\_ptr()} and throw a specific exception.

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

6.3.2 Conversion from \(T\)

An object of type \(T\) is convertible to an expected object of type \texttt{expected<T,E>}:

\[
\texttt{expected<int, error\_condition> ei} = 1; // works}
\]

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

\[
\texttt{expected<int, error\_condition> ei\{in\_place, 1\};}
\]

If the latter appears too cumbersome, one can always use function \texttt{make\_expected} described below:

\[
\texttt{expected<int> ei = make\_expected(1);}
\]
\[
\texttt{auto ej = make\_expected(1);}
\]
6.3.3 Conversion from E

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

```cpp
expected<string, int> exp1 = make_unexpected(1);
expected<string, int> 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:

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

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<T,E> (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.

6.4 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.

6.4.1 Explicit conversion to bool

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

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

6.4.2 Accessing the contained value

Even if expected<T,E> 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<T,E>. 
6.4.3 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:

```plaintext
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.

6.4.4 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.

```plaintext
void interact() {
    string s;
    cout << "enter number: ";
    cin >> s;
    expected<int, error> ei = str2int(s);

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

The exception thrown 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<T>`, it throws the exception stored in the `exception_ptr`. An approach enabling customization of this behavior is presented in the section 8.2.

`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.

6.4.5 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)`.

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

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

6.4.6 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:

\[ f\.get\_unexpected() == \text{make\_unexpected}(f\.error()); \]
\[ f\.get\_unexpected() == \text{expected}\{T, E\}(\text{unexpect}, f\.error()); \]

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

```cpp
expected<pair<int, int>, errc> getIntRange(istream_range& r) {
  auto f = getInt(r);
  if (!f) return f.get_unexpected();

  auto m = matchedString("..", r);
  if (!m) return m.get_unexpected();

  auto l = getInt(r);
  if (!l) return l.get_unexpected();

  return std::make_pair(*f, *l);
}
```

get_unexpected is also provided for symmetry purpose. On one side, there is an implicit conversion from unexpected<E> to expected<T,E> and on the other side there is an explicit conversion from expected<T,E> to unexpected<E>. A more pleasant function manipulating error is catch_error(F) and is explained in the monadic operations in section 6.9.

6.4.7 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:

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

6.4.8 Relational operators

As optional, one of the design goals of expected is that objects of type expected<T,E> should be valid elements in STL containers and usable with STL algorithms (at least if objects of type T and E are). Equality comparison is essential for expected<T,E> to model concept Regular. C++ does not have concepts, but being regular is still essential for the type to be effectively used with STL. Ordering is essential if we want to store expected values in ordered associative containers. A number of ways of including the unexpected state in comparisons have been suggested. The ones proposed, have been crafted such that the axioms of equivalence and strict weak ordering are preserved: unexpected values stored in expected<T,E> are simply treated as additional values that are always different from T; these values are always compared as less than any value of T when stored in an expected object.

The main issue is how to compare the unexpected values between them. operator==() is defined for exception_ptr, using shallow semantics but there is no order between two exception_ptr.

```cpp
template <class T, class E>
constexpr bool operator<(const expected<T,E>& x, const expected<T,E>& y) {
  return (x)
    ? (y) ? *x < *y : false
    : (y) ? true : ?<?;
}
```

```cpp
template <class T, class E>
constexpr bool operator==(const expected<T,E>& x, const expected<T,E>& y) {
  return (x)
    ? (y) ? *x == *y : false
    : (y) ? false : ?==?;
}
```
If we follow the `optional<T>` semantics, two unexpected values should always be equal and do not compare. That is, `<?>` should be substituted by `false` and `?==?` by `true`. However considering all the unexpected value equals seems counterintuitive.

The alternative consists in forwarding the request to the respective `unexpected_type<E>` relational operators. That is, `<?>` should be substituted by `x.get_unexpected() < y.get_unexpected()` and `?==?` by `x.get_unexpected() == y.get_unexpected()`.

But how to define the relational operators for `unexpected_type<E>`? We can forward the request to the respective `E` relational operators when `E` defines these operators and follows the `optional<T>` semantics otherwise.

The case of `unexpected_type<std::exception_ptr>` could follow the `optional<T>` semantics as the shallow comparison is not very useful.

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

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

Unexpected values could have been as well considered greater than any value of `T`. The choice is a great degree arbitrary. We choose to stick to what `std::optional` does.

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

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

Although it is difficult to imagine any practical use case of ordering relation between `expected<T,E>` and `unexpected_type<E>`, we still provide it for completeness sake.

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

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

Given that `expected<T,E>` is comparable and implicitly constructible from `T`, the mixed comparison is there already. We would have to artificially create the mixed overloads only for them to cause controlled compilation errors. A consistent approach to prohibiting mixed relational operators would be to also prohibit the conversion from `T` or to also prohibit homogenous relational operators for `expected<T,E>`; we do not want to do either, for other reasons discussed in this proposal. Also, mixed relational operations are available in `std::optional<T>` and we want to maintain the same behavior for `expected<T,nullopt_t>` and `optional<T>`. Mixed operators come as something natural when we consider the model "T with additional values".

For completeness sake, we also provide ordering relations between `expected<T,E>` and `unexpected_type<E>`, even though we see no practical use case for them:

```cpp
bool test(expected<unsigned, int> e)
{
    assert (e >= make_unexpected(1));
}
assert (!(e < make_unexpected(1)));  
assert (make_unexpected(1) <= e);  
return (e > make_unexpected(1));

There exist two ways of implementing operator>() for expected objects: use T::operator>() or use
expected<T,E>::operator<()  
In case T::operator> and T::operator< are defined consistently, both above implementations are equivalent.  
If the two operators are not consistent, the choice of implementation makes a difference.  
For relational operations, we choose to implement all in terms of expected<T,E>::operator<() to be consistent  
with the choice taken for std::optional.

The same applies to the relational operators for unexpected_type<E>.

6.5 Modifiers

6.5.1 Resetting the value

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

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

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

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

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

6.5.5 Expected references

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

6.5.6 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.
6.6 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.

6.7 Moved from state

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

6.8 IO operations

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

6.9 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 4.1 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.

6.9.1 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<T,E>.

- (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.

6.9.2 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 8.1, 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.

6.9.3 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.
6.9.4 Exception thrown in the continuation

This behavior is left open in the section 8.2. Currently, the exceptions thrown in the continuations are not caught.

6.9.5 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.

```cpp
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.

```cpp
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.

6.9.6 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:

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

7 Related types

7.1 Variant

expected<T,E> 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<T,E> is more specific and closer to what the user could expect as the result type of a function. In addition, expected<T,E> doesn’t intend to be used to define recursive data as boost::variant<> does.

The table 2 presents a brief comparison between boost::variant<unexpected<E>, T> and expected<T,E>.
<table>
<thead>
<tr>
<th>Feature</th>
<th>boost::variant&lt;unexpected&lt;E&gt;, T&gt;</th>
<th>expected&lt;T,E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>never-empty warranty</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>accepts is_same&lt;T,E&gt;</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>swap</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>factories</td>
<td>no</td>
<td>make_expected / make_unexpected</td>
</tr>
<tr>
<td>hash</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>value_type</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>default constructor</td>
<td>yes (if T is default constructible)</td>
<td>yes (if T is default constructible)</td>
</tr>
<tr>
<td>observers</td>
<td>boost::get&lt;T&gt; and boost::get&lt;E&gt;</td>
<td>pointer-like / value / error / value_or</td>
</tr>
<tr>
<td>continuations</td>
<td>apply_visitor</td>
<td>map/bind/then/catch_error</td>
</tr>
</tbody>
</table>

Table 2: Comparison between variant and expected.

7.2 Optional

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

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

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

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

7.3 Promise and Future

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

```cpp
template <class T>
expected<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<T>`.

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

where make_unexpected_future is defined as:

```cpp
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:

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

As for the `future<T>` proposal, `expected<T,E>` 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<T,E>`. This make it easier to reuse functions.

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

### 7.4 Comparison between optional, expected and future

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

### 8 Open points

#### 8.1 Better support for monad

In the use-cases section (4.1), 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:

```cpp
expected<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.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Optional</th>
<th>Expected</th>
<th>Promise/Future</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>specific null value</strong></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>relational operators</strong></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>swap</strong></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>factories</strong></td>
<td>make_optional / nullopt</td>
<td>make_expected / make_unexpected</td>
<td>make_ready_future / (make_exceptional, see [6])</td>
</tr>
<tr>
<td><strong>hash</strong></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>value_type</strong></td>
<td>yes</td>
<td>yes</td>
<td>no / (yes, see [6])</td>
</tr>
<tr>
<td><strong>default constructor</strong></td>
<td>yes</td>
<td>yes (if T is default constructible)</td>
<td>yes</td>
</tr>
<tr>
<td><strong>allocators</strong></td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>emplace</strong></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>bool conversion</strong></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>state</strong></td>
<td>bool()</td>
<td>bool() / valid</td>
<td>valid / ready / (has_value, see [6])</td>
</tr>
<tr>
<td><strong>observers</strong></td>
<td>pointer-like / value / value_or</td>
<td>pointer-like / value / error / value_or</td>
<td>get / (get_exception_ptr, see [6])</td>
</tr>
<tr>
<td><strong>visitation</strong></td>
<td>no</td>
<td>map / bind / then / catch_error</td>
<td>then / (next/recover see [6])</td>
</tr>
<tr>
<td><strong>grouping</strong></td>
<td>n/a</td>
<td>n/a</td>
<td>when_all / when_any</td>
</tr>
</tbody>
</table>

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

```cpp
expected<int, error_condition> 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:

```cpp
expected<int, error_condition> 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

```cpp
expected<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):

```cpp
expected<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.
8.2 A Configurable Expected

Expected might be configurable through a trait expected_traits. The first variation point is the behavior of value() when expected<T,E> contains an error. The current strategy throw a bad_expected_access exception (or the contained exception if the type is expected<T, exception_ptr>) 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<T, E>, 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<int, error_condition> f(int i, const expected<int, error_condition>& 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<T, error_condition>>
   {
   static expected<T, error_condition> 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.

8.3 Allocator support

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

typedef vector<expected<vector<int, MyAlloc>, error>, 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.
8.4 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,

```cpp
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`:

```cpp
e = f();
auto i = e.value_or_throw<std::system_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:

```cpp
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:

```cpp
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 throw:

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

that could be used as follows:

```cpp
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.
8.5 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, 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.

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

This proposal makes expected<T,exception_ptr> equality comparable making all the unexpected values equals as exception_ptr equality comparison is shallow and doesn’t provides relational operators.

Should expected<T,exception_ptr> not be equality comparable?

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

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

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

As for Rapperswil decision, the unexpected values of expected<T,E> when E is not exception_ptr don’t collapse to a unique value.

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

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

8.9 Should expected<T,E> 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?

8.10 Should a specific exception be thrown when the expected<T> doesn’t have a value neither an exception stored?

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

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

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

It is easy to catch the exceptions when the type is expected<T>. However, doing it for expected<T,E> 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<T>::map/bind/then catch the exceptions and propagate them on the result?

Should expected<T,E>::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?
8.12 Do we need a expected<T,E>::error_or function?

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

\[ \text{expected<T,E>} \] would be seen more like something

\[
\text{struct E; optional<T> .}
\]

The following code was show as use case

\[
\text{auto e = function();}
\]

\[
\text{switch (e.status())}
\]

\[
\text{success: ....; break; too_green: ....; break; too_pink: ....; break;}
\]

With the current interface the user could be tempted to do

\[
\text{auto e = function();}
\]

\[
\text{if (e)}
\]

\[
\text{/*success:*/ ....;}
\]

\[
\text{else}
\]

\[
\text{switch (e.status())}
\]

\[
\text{case too_green: ....; break; case too_pink: ....; break;}
\]

This could be done with the current interface as follows

\[
\text{auto e = function();}
\]

\[
\text{switch (error_or(e, success))}
\]

\[
\text{success: ....; break; too_green: ....; break; too_pink: ....; break;}
\]

where

\[
\text{template <class E, class T>}
\]

\[
E \text{ error_or(expected<T,E> const& , E err) } \{
\text{if(e) return err; else return error();}
\}
\]

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

8.13 Do we need a expected<T,E>::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.

\[
\text{e = function();}
\]

\[
\text{while ( e.status == timeout )}
\]

\[
\text{sleep(delay); delay *=2;}
\]

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

\[
\text{e = function();}
\]

\[
\text{while ( has_error(e, timeout) )}
\]

\[
\text{sleep(delay); delay *=2;}
\]
where

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

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

9 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

X.Y.1 In 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`

```cpp
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 explicit unexpected_type<decay_t<E>> make_unexpected(E&& v);
            unexpected_type<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

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

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

**Effects:**

Build an unexpected by copying the parameter to the internal storage `val`.
constexpr explicit unexpected_type(E &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

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

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

X.Y.6 In general
This subclause describes class template expected that represents expected objects. An \texttt{expected<T,E> object}

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.

\textbf{X.Y.7 Header \langle experimental/expected\rangle synopsis} [expected.synop]

```cpp
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<void, E>;

    // X.Y.10, Specialization of expected as a meta-function : \( T \rightarrow \texttt{expected<T,E>} \).
    template <class E>
    class expected<holder, E>;

    // X.Y.12, unexpect tag
    struct unexpect_t();
    constexpr unexpect_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<T,E>&, const expected<T,E>&);
    template <class T, class E>
    constexpr bool operator!= (const expected<T,E>&, const expected<T,E>&);
    template <class T, class E>
    constexpr bool operator<(const expected<T,E>&, const expected<T,E>&);
    template <class T, class E>
    constexpr bool operator>(const expected<T,E>&, const expected<T,E>&);
    template <class T, class E>
    constexpr bool operator<=(const expected<T,E>&, const expected<T,E>&);
    template <class T, class E>
    constexpr bool operator>=(const expected<T,E>&, const expected<T,E>&);

    // X.Y.15, Comparison with \( T \)

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

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

// X.Y.16, Comparison with unexpected_type<E>

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

// X.Y.17, Specialized algorithms

void swap(expected<E,T>&, expected<E,T>&) noexcept(see below);
void swap(expected<T,E>&, expected<T,E>&) noexcept(see below);

// X.Y.18, Factories

template <class T> constexpr expected<decay_t<T>> make_expected(T&& v);
expected<void> make_expected();
template <class E> expected<void, E> make_expected();

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

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

template <class F>
constexpr expected<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<T,E>>;

}}}

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

X.Y.8 Definitions [expected.defs]
An instance of `expected<T,E>` is said to be valued if it contains an value of type `T`. An instance of `expected<T,E>` is said to be unexpected if it contains an object of type `E`.

X.Y.9 expected for object types

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

        template <class U>
        struct rebind {
            typedef expected<U, error_type> 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;

// X.Y.9.6, factories

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&;

// X.Y.9.6, factories

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

template <typename F>
expected<decltype(func(declval<T>())),E> map(F&& func);

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

Valued instances of expected<T,E> where T and E is of object type shall contain a value of type T or a value of type E within its own storage. This value is referred to as the contained or the unexpected value of the expected object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained or unexpected value. The contained or unexpected value shall be allocated in a region of the expected<T,E> storage suitably aligned for the type T and E.

Members has_value, val and err are provided for exposition only. Implementations need not provide those members. has_value indicates whether the expected object’s contained value has been initialized (and not yet destroyed); when has_value is true val points to the contained value, and when it is false err points to the erroneous value.

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

X.Y.9.1 Constructors [expected.object.ctor]

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

Effects:
Initializes the contained value as if direct-non-list-initializing an object of type T with the expression T0.
```
Postconditions:
  bool(*this).
Throws:
  Any exception thrown by the default constructor of T.
Remarks:
  The expression inside noexcept is equivalent to:
  is_nothrow_default_constructible<T>::value.
Remarks:
  This signature shall not participate in overload resolution unless
  is_default_constructible<T>::value.

expected(const expected& rhs);
Effects:
If bool(rhs) initializes the contained value as if direct-non-list-initializing an object of type T with the expression *rhs.
If !bool(rhs) initializes the contained value as if direct-non-list-initializing an object of type E with the expression rhs.error().
Postconditions:
  bool(rhs) == bool(*this).
Throws:
  Any exception thrown by the selected constructor of T or E.
Remarks:
  This signature shall not participate in overload resolution unless
  is_copy_constructible<T>::value and
  is_copy_constructible<E>::value.

expected(expected && rhs) noexcept('see below');
Effects:
If bool(rhs) initializes the contained value as if direct-non-list-initializing an object of type T with the expression std::move(*rhs).
If !bool(rhs) initializes the contained value as if direct-non-list-initializing an object of type E with the expression std::move(rhs.error()).
Postconditions:
  bool(rhs) == bool(*this) and
  bool(rhs) is unchanged.
Throws:
  Any exception thrown by the selected constructor of T or E.
Remarks:
  The expression inside noexcept is equivalent to:
  is_nothrow_move_constructible<T>::value == 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(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`.

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

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

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

**Throws:**
- Any exception thrown by the selected constructor of \( T \).

**Remarks:**
- If \( T \)'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```cpp
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_move_constructible<T>::value`.

```cpp
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`.

**Remarks:**
- If \( T \)'s constructor selected for the initialization is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.
This signature shall not participate in overload resolution unless
\(\text{is\_constructible}\langle T, \text{initializer\_list}\langle U\rangle&, \text{Args}\&\ldots\rangle::\text{value} \). 

\texttt{constexpr expected(unexpected\_type\langle E\rangle \text{ const} & e);}

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

\textbf{Postconditions:}
- \(! *\text{this}\).

\textbf{Throws:}
- Any exception thrown by the selected constructor of \(E\).

\textbf{Remarks:}
- If \(E\)'s selected constructor is a \texttt{constexpr} constructor, this constructor shall be a \texttt{constexpr} constructor.

\textbf{Remarks:}
- This signature shall not participate in overload resolution unless \(\text{is\_move\_constructible}\langle E\rangle::\text{value} \).

\texttt{constexpr expected(unexpected\_type\langle E\rangle \&\& e);}

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

\textbf{Postconditions:}
- \(! *\text{this}\).

\textbf{Throws:}
- Any exception thrown by the selected constructor of \(E\).

\textbf{Remarks:}
- If \(E\)'s selected constructor is a \texttt{constexpr} constructor, this constructor shall be a \texttt{constexpr} constructor.

\textbf{Remarks:}
- This signature shall not participate in overload resolution unless \(\text{is\_copy\_constructible}\langle E\rangle::\text{value} \).

\textbf{X.Y.9.2 Destructor} \[\text{expected.object.dtor}\]

\texttt{-expected();}

\textbf{Effects:}
- If \(\text{is\_trivially\_destructible}\langle T\rangle::\text{value} != \text{true}\) and \(! \text{bool}(\ast\text{this})\), calls \(\text{val}\rightarrow T::\text{-}T()\).
- If \(\text{is\_trivially\_destructible}\langle E\rangle::\text{value} != \text{true}\) and \(! \ast\text{this}\), calls \(\text{err}\rightarrow E::\text{-}E()\).

\textbf{Remarks:}
- If \(\text{is\_trivially\_destructible}\langle T\rangle::\text{value} \) and \(\text{is\_trivially\_destructible}\langle E\rangle::\text{value} \) then this destructor shall be a trivial destructor.

\textbf{X.Y.9.3 Assignment} \[\text{expected.object.assign}\]

\texttt{expected<T,E>& operator=(\text{const expected<T,E>&} rhs);} 

\textbf{Effects:}
- If \(\text{bool}(\ast\text{this})\) and \(\text{bool}(\text{rhs})\), assigns \(\text{rhs}\) to the contained value \(\text{val}\), otherwise
- If \(\text{bool}(\ast\text{this})\) and \(! \text{rhs}\), destroys the contained value by calling \(\text{val}\rightarrow T::\text{-}T()\) and initializes the contained value as if direct-non-list-initializing an object of type \(E\) with \(\text{rhs}\.\text{error}()\), otherwise
- If \(! \ast\text{this} \) and \(\text{rhs}\), assigns \(\text{rhs}\.\text{error}()\) to the contained value \(\text{err}\), otherwise
- If \(! \ast\text{this} \) and \(\text{bool}(\text{rhs})\), destroys the contained value by calling \(\text{err}\rightarrow E::\text{-}E()\) and initializes the contained value as if direct-non-list-initializing an object of type \(E\) with \(\text{rhs}\.\text{error}()\).

\textbf{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<T,E>& operator=(expected<T,E>&& 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, 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(), 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<T,E>& 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:

32
*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<T,E>& 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.

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

**Effects:**

if ! *this assigns std::forward<E>(e.value()) to the contained value 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... Args>
void emplace(Args&&... args);

**Effects:**

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

**Postconditions:**

bool(*this).

**Exception Safety:**

If an exception is thrown during the call to T's 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 emplace(initializer_list<U> il, Args&&... args);

**Effects:**
if bool(*this), assigns the contained value \( \text{val} \) as if constructing an object of type \( T \) with the arguments \( \text{il, std::forward<Args>(args)} \ldots \), otherwise destroys the contained value by calling \( \text{err->E::~E()} \) and initializes the contained value as if constructing an object of type \( T \) with the arguments \( \text{il, std::forward<Args>(args)} \ldots \).

**Postconditions:**
bool(*this).

**Exception Safety:**
If an exception is thrown during the call to \( T \)'s constructor, ! *this, and the previous \( \text{val} \) (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:

\[ \text{is_constructible<T, initializer_list<U>&, Args&&...>::value} \]

---

**X.Y.9.4 Swap**

\[ \text{expected.object.swap} \]

```c
void swap(expected<T,E>& 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:

\[ \text{is_nothrow_move_constructible<T>::value && noexcept(swap(declval<T&>(), declval<T&>())) && } \]

\[ \text{is_nothrow_move_constructible<E>::value && noexcept(swap(declval<E&>(), declval<E&>()))}. \]

**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**

\[ \text{expected.object.observe} \]

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

**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.

```c
constexpr T const& operator *() const&;
T& operator *() &;
```
Requires:
  bool(*this).

Returns:
  val.

Remarks:
  The first function shall be a constexpr function.

constexpr T&& operator *() &&;

Requires:
  bool(*this).

Returns:
  move(val).

Remarks:
  This function shall be a constexpr function.

constexpr explicit operator bool() noexcept;

Returns:
  has_value.

Remarks:
  This function shall be a constexpr function.

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

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

Throws:
  • When E is std::exception_ptr as if throw_exception(error()) if !*this
  • Otherwise bad_expected_access(err) if !*this.

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

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

Requires:
  !*this.

Returns:
  err.

Remarks:
  The first function shall be a constexpr function.

constexpr E&& error() &&;

Requires:
  !*this.

Returns:
  move(err).

Remarks:
  The first function shall be a constexpr function.

template <class Ex>
bool expected<T>:::has_exception() const;
Returns:

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

```cpp
constexpr unexpected<E> get_unexpected() const;
```

Requires:

`!(this)`.

Returns:

`make_unexpected(err)`.

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

Returns:

`bool(*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.

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

Returns:

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

Exception Safety:

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

The function shall not participate in overload resolution unless:

- `is_copy_constructible<T>::value`
- `is_convertible<U&&, T>::value`.

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

Returns:

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

Throws:

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

Remarks:
The function shall not participate in overload resolution unless:

is_copy_constructible<expected<T,E>>::value

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

Returns:
*this.

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

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

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

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

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

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

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

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

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

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

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

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

Effects:
if has_exception<Ex>() call the continuation function funct with the stored exception as parameter.

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

template <class Ex, class F>
'see below' map(F&& func)

Returns:
if bool(*this) returns returns expected<decltype(func(move(val))), E>(func(move(val))), otherwise, returns get_unexpected().
template <class Ex, class F> 'see below' bind(F& func)

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

template <class Ex, class F> 'see below' then(F& func);

Returns:
returns unwrap(expected<decltype(func(move(*this))), E>(func(move(*this)))).

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

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

X.Y.10 expected as a meta-function

template <class E>
class expected<holder, E>

public:
  template <class T>
    using type = expected<T, E>
;

X.Y.11 expected for void

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

    template <class U>
    struct rebind {
      typedef expected<U, error_type> type;
    };

    // ??, constructors
    constexpr expected() noexcept;
    expected(const expected&);
    expected(expected&&) noexcept(see below);
    constexpr explicit expected(in_place_t);
    constexpr expected(unexpected_type<E> const&);
    template <class Err>
    constexpr expected(unexpected_type<Err> const&);

    // ??, destructor
    -expected();

    // ??, assignment
expected& operator=(const expected&);
expected& operator=(expected&&) noexcept(see below);
void emplace();

// ??, swap
void swap(expected&) noexcept(see below);

// ??, observers
constexpr explicit operator bool() const noexcept;
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<void,E> catch_exception(F&& f);

template <typename F>
expected<decltype(func()), E> 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.

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

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

```cpp
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

TODO: Describe the functions.

### X.Y.15 Comparison with $\tau$

TODO: Describe the functions.

### X.Y.16 Comparison with `unexpected<E>`

TODO: Describe the functions.

### X.Y.17 Specialized algorithms

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

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

### X.Y.18 Expected Factories

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

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

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

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

```cpp
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)))`.

```cpp
template <class T, class E>
constexpr expected<decay_t<E>,T> make_expected_from_error(E e);
```
Returns:
\[
\text{expected<decay_t<E>,T>(make_unexpected(e))};
\]

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

Returns:
\[
\text{expected<exception_ptr,T>(make_unexpected_from_current_exception())}
\]

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

Equivalent to:
\[
\begin{align*}
\text{try} \\
\text{return make_expected(funct());} \\
\text{catch (...) return make_unexpected_from_current_exception();}
\end{align*}
\]

X.Y.19 Hash support

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

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<T,E>> shall meet the requirements of class template hash. For an object e of type expected<T,E>, if bool(e), hash<expected<T,E>>(e) shall evaluate to 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<void, E>>;

Requires:

10 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].

11 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 KrzemieÅ„ski 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


