Document number: N4015
Date: 2014-05-20
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>
Pierre Talbot <ptalbot@hyc.io>

A proposal to add a utility class to represent expected monad

Contents

1 Introduction

Class template `expected<E,T>` proposed here is a type that may contain a value of type `T` or a value of type `E` in its storage space. `T` represents the expected value, `E` represents the reason explaining why it doesn't contains a value of type `T`, that is the unexpected value. Its interface allows to query if the underlying value is either the expected value (of type `T`) or an unexpected value (of type `E`). The original idea comes from Andrei Alexandrescu C++ and Beyond 2012: Systematic Error Handling in C++ talk [**?**]. The interface and the rational are based on `std::optional` N3793 [**?**] and Haskell monads. We can consider that `expected<E,T>` is a generalization of `optional<T>` providing in addition a monad interface and some specific functions associated to the unexpected type `E`. It requires no changes to core language, and breaks no existing code.

2 Motivation and Scope

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

- **Error visibility** Failure cases should appears throughout the code review. Because the debug can be painful if the errors are hidden.
- **Information on errors** The errors should carry out as most as possible information from their origin, causes and possibly the ways to resolve it.

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

Table 1: Comparison between two error handling systems.

- **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.

2.1 Alexandrescu Expected class

We can do the same analysis for the `Expected<T>` class from Alexandrescu talk [**?**]:

- **Error visibility** It takes the best of the exception and error code. It's visible because the return type is Expected<T> and if the user ignore the error case, it throws the contained exception.
- **Information** As rich as exception.
- **Clean code** It's up to the programmer to choose handling errors as error code or to throw the contained exception.
- **Non-Intrusive** Use the return channel without monopolizing it.

Other characteristics of `Expected<T>`:

- Associates errors with computational goals.
- Naturally allows multiple exceptions inflight.
- Switch between "error handling" and "exception throwing" styles.
- Teleportation possible.
  - Across thread boundaries.
  - Across nothrow subsystem boundaries.
  - Across time: save now, throw later.
- Collect, group, combine exceptions.

However `Expected<T>` class also has some minor limitations:

- The error code must be an exception.
- It doesn't provide a better solution to resolve errors. You can throw or use the `hasException<E>()` function to test errors which is similar to the old switch case statement.
- The function `hasException<E>()` test the type and so cannot distinguish two different errors from the same exception. Exception can contains multiple error case scenarios (think about `std::invalid_argument`).

2.2 Differences between the proposed expected class and Alexandrescu Expected class

The main enhancements or differences of the proposed `expected<E,T>` respect to `Expected<T>` are:

- `expected<E,T>` parameterizes the root cause that prevents its creation, `expected<E,T>` is either a `T` or the root cause `E` that prevents its creation.
- `expected<E,T>` is default constructible.
- `expected<E,T>` is implicitly constructible from an `unexpected_type<E>`.
- `expected<E,T>` is a monad error (see [**?**]).

3 Use cases

3.1 Safe division

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

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

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

Which, using `expected<exception_ptr,int>`, turns to:

```
expected<exception_ptr,int> safe_divide(int i, int j)
{
  if (j==0) return make_unexpected(DivideByZero()); // (1)
  else return i / j; // (2)
}
```

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

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

becomes using `expected<exception_ptr,int>`:

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

However `expected<E,T>` delivers cleaner code when used in a functional style:

```
expected<exception_ptr,int> f1(int i, int j, int k)
{
  return safe_divide(j, k).fmap([&](int q){
    return i + q;
  });
}
```

The `fmap` members 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 an example using functor:

```
expected<exception_ptr,int> f1(int i, int j, int k)
{
  return safe_divide(j, k).fmap(bind(plus, i, _1));
}
```

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

```
struct NotDivisible: public std::exception
{
  int i, j;
  NotDivisible(int i, int j) : i(i), j(j) {}
};

expected<exception_ptr,int> safe_divide(int i, int j)
{
  if (j == 0) return make_unexpected(DivideByZero());
  if (i%j != 0) return make_unexpected(NotDivisible(i,j));
  else return i / j;
}
```

Now we have a division function for integers that possibly fail in two ways. However, it's not easy to write code that detect which of the two conditions occurred. For instance, we might have situations where dividing two integers which are not evenly divisible is OK (we just throw away the remainder) but division by zero is probably never going to be OK. Let's try to write this using our safe_divide function, first using exceptions:

```
T divide(T i, T j)
{
  try
  {
    return safe_divide(i,j)
  }
  catch(NotDivisible& ex)
  {
    return ex.i/ex.j;
  }
  catch(...)
  {
    throw;
  }
}
```

and then using `expected<E,T>`:

```
expected<exception_ptr,int> divide(int i, int j)
{
  auto e = safe_divide(i,j);
  if(e.has_exception<NotDivisible>())
    return i/j;
  else
    return e;
}
```

The `has_exception` function throws the contained exception and thus should not be called multiple times to discriminate over exceptions in an if-else statement if we care about performances. A more efficient way is shown in section 3.2.1.

```
expected<exception_ptr,int> divide(int i, int j)
{
```

```
  return safe_divide(i,j).catch_excception<NotDivisible>([](auto &ex)
    return ex.i/ex.j;
  });
}
```

Lets continue with the exception-oriented function $i/k + j/k$:

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

Now let's write this code using an `expected<E,T>` type and an imperative flavour:

```
expected<exception_ptr,int> f2(int i, int j, int k)
{
  auto q1 = safe_divide(i, k);
  if (!q1) return q1;

  auto q2 = safe_divide(j, k);
  if (!q2) return q2;

  return *q1 + *q2;
}
```

This is nice in the sense that whenever there is an error we get a specific error result. However the "clean code" characteristic introduced in section 2 is not well-respected as the readability doesn't differ much from the "C return code". We can rewrite this using a functional style using the member function `mbind`[1]:as follows:

```
expected<exception_ptr,int> f(int i, int j, int k)
{
  return safe_divide(i, k).mbind([=](int q1) {
      return safe_divide(j,k).fmap([=](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. A cleaner solution uses the variadic free function `fmap`[2]:

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

The function `fmap` returns the first erroneous expected argument or, if they all contain a value, the result of the `plus` operation.

Now let's rewrite this using a possible C++ language extension: adding a DO expression like the Haskell do expression. It is something similar to the `await` extension for futures however it is not limited to futures. It could works for any monad.

The grammar could be

```
do-expression ::= do-initialization ':' do-expression-or-expression

do-expression-or-expression ::= do-expression | expression

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

The meaning of `do-expression` is given by a transformation `[[ ]]`

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

[1] `mbind` stands for "monadic bind".
[2] `fmap` stands for "functor map".

The previous function could be written as

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

resulting in the transformed C++ code

```
expected<exception_ptr,int> f2(int i, int j, int k)
{
  return mbind(safe_divide(i, k) ,[&r](auto s1) {
    return mbind(safe_divide(j, k),[&r](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 (see [**?**] and define `f2` as

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

In the case of expected and optional, that is, monads that are always ready and have only one value stored, the following macro

```
#define EXPECT(V, EXPR) \
auto BOOST_JOIN(expected,V) = EXPR; \
if (! has_value(BOOST_JOIN(expected,V))) return get_unexpected(BOOST_JOIN(expected,V)); \
auto V = deref(BOOST_JOIN(expected,V))
```

can be used to obtain the same result.

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

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

3.2 Error retrieval and correction

The major advantage of `expected<E,T>` over `optional<T>` is the ability to transport un 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 behaviour 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.

3.2.1 Exception-based expected

The following example (previously introduced in section 3.1) shows how to transform an expected interface into the corresponding exception one's. It is useful if we want to switch over different error-handling style.

```
int f2(int i, int j, int k)
{
  // if the underlying expected doesn't contain a value
  // it throws the contained exception.
  int q1 = safe_divide(i, k).value();
  int q2 = safe_divide(j, k).value();

  return q1 + q2;
}
```

In some cases, even if the underlying error system is exception-based, we don't want to throw it away but to directly recover from the unexpected value.

```
int divide_or_0(int i, int j) noexcept
{
  auto e = safe_divide(i,j);
  return *(e.catch_error(const exception_ptr& e) {
    return 0;
  }));
}
```

The `catch_error` function calls the continuation if the expected is erroneous. In some case, we'll be able to recover only from a subset of the possible error. With exception, the most efficient way is to throw the contained exception and catch the ones we're able to recover from.

```
expected<exception_ptr,int> divide_lower_bound(int i, int j)
{
  auto e = safe_divide(i,j);
  return e.catch_error(const exception_ptr& e) {
    try
    {
      rethrow_exception(e);
    }
    // If it failed because it wasn't an integer division.
    catch(const NotDivisible& d)
    {
      return d.i/d.j;
    }
    catch(...)
    {
      return make_unexpected(e);
    }
  });
}
```

Some might argue that this solution is nearly identical to the one with exception-only presented in section 3.1. The main advantages are the error treatment isolation (it can be encapsulated in a function) and we keep the basic advantages of expected despite manipulating exception.

A better alternative in this case is to use the catch_exception member function

```
expected<exception_ptr,int> divide_lower_bound(int i, int j)
{
  return safe_divide(i,j)
  .catch_exception<NotDivisible>(auto& d) // If it failed because it wasn't divisible.
    {
      return d.i/d.j;
    }
  );
}
```

3.2.2 Error-based expected

The two last examples would work similarly with any other kinds of error. For example here we're using the `error_condition` type.

When the `Error` template parameter is not `std::exception_ptr`, the functions `has_exception()` and `catch_exception()` have no sense. In this case the user could use the member function `error()`).

```
expected<error_condition, int> divide_lower_bound(int i, int j)
{
  auto e = safe_divide(i,j);
  if ( !e && e.error() == divide_err::not_divisible ) {
    return i/j;
  }
  return e;
}
```

The member function `catch_error` could be used with a continuations passing style to try to recover from an error.

```
expected<error_condition, int> divide_lower_bound(int i, int j)
{
  auto e = safe_divide(i,j);
  return e.catch_error([i,j](const error_condition& e){
    if(e.value() == divide_err::not_divisible)
    {
      return i/j;
    }
    else
    {
      return make_unexpected(e);
    }
  });
}
```

The code is similar to the one with exceptions but do not suffer from performance issue since we don't re-throw exceptions.

As `expected<error_condition, int>` doesn't store a exception, when it doesn't contains a value the member function `value` throws a `bad_expected_access` exception wrapping the stored error.

4 Impacts on the Standard

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

5 Design rationale

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

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

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

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

Additionally, within the affordable limits, we propose the view that `expected<E,T>` extends the set of the values of `T` by the values of type `E`. This is reflected in initialization, assignment, ordering, and equality comparison with both `T` and `E`. In the case of `optional<T>`, `T` can not be a `nullopt_t`. As the types `T` and `E` could be the same in `expected<E,T>`, there is need to tag the values of `E` to avoid ambiguous expressions. The

`make_unexpected(E)` function is proposed for this purpose. However `T` can not be `unexpected_type<E>` for a given `E`.

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

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

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

5.2 Initialization of `expected<E,T>`

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

As in [**?**], the model retained is to initialize either by providing either an already constructed `T` or a tagged `E`.

```
string s{"STR"};

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

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

In order to create a unexpected object a special function needs to be used: `make_unexpected`:

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

As in [**?**], and in order to avoid calling move/copy constructor of `T`, we use a 'tagged' placement constructor:

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

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

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

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

The alternative and also comprehensive initialization approach, which is not compatible with the choice to default construct `expected<E,T>` to `E()`, could be to have a variadic perfect forwarding constructor that just forwards any set of arguments to the constructor of the contained object of type `T`.

5.3 Almost Never-empty guaranty

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

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

5.3.1 The default constructor

- `std::experimental::optional<T>` default constructs to an optional with no value.
- `boost::variant<T,E>` default constructs to a variant with the value `T()` it `T` is default constructible or to the value `E()` it `E` is default constructible or it is ill formed otherwise.
- `std::future<T>` default constructs to an invalid future with no shared state associated, that is, no value and no exception.
- `std::experimental::optional<T>` default constructor is equivalent to `boost::variant<nullopt_t, T>`.
- Should the default constructor of `std::experimental::expected<E,T>` behave like `boost::variant<T,E>` or as `boost::variant<E,T>`?
- Should the default constructor of `std::experimental::expected<E,T>` behave like `std::experimental::optional<boost::v`
- Should the default constructor of `std::experimental::expected<nullopt_t,T>` behave like `std::experimental::optional<` If yes, how should behave the default constructor of `std::experimental::expected<E,T>`? as if initialized with `make_unexpected(E())`? This will be equivalent to the initialization of `boost::variant<E,T>`.
- Should `std::experimental::expected<E,T>` provide a default constructor at all? [**?**] present valid arguments against this approach, e.g. `array<optional<T>>` would not be possible.

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

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

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

5.3.2 Conversion from `T`

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

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

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

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

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

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

5.3.3 Using `make_unexpected` to convert from `E`

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

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

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

exp1 =   make_unexpected(1);
exp2 =  {unexpect, 1};
```

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

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

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

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

The usage of `make_unexpected` is also a consequence of the adapted model for expected: a discriminated union of `T` and `unexpected_type<E>`. While `make_unexpected(E)` has been chosen because it clearly indicates that we are interested in creating an unexpected `expected<E,T>` (of unspecified type `T`), it could be used also to make a ready future with a specific exception, but this is outside the scope of this proposal.

Note also that the definition of the result type of `make_unexpected` has explicitly deleted default constructor. This is in order to enable the reset idiom

```
exp2 = {};
```

which would otherwise not work because of ambiguity when deducing the right-hand side argument.

5.3.4 Why not a `make_unexpected` nested function on `expected<E,T>`?

[**?**] `Expected<T>` class has a nested member function `Expected<T>::from_error(E)` instead of the free function `make_unexpected`. But the proposed `expected<E,T>` has an additional template parameter `E` and so the type `E` would be explicit.

```
expected<unsigned, string> ei = expected<unsigned, string>::make_unexpected(1);
```

This has however several advantages. Namespace `std` is not polluted with an additional expected-specific name. Also, it resolves certain ambiguities when types like expected<E, expected<E,T» are involved:

```
expected<int, expected<int, string>> eei =
    expected<int, string>::make_unexpected(1);          // valued
expected<int, expected<int, string>> eej =
    expected<int, expected<int, string>>::make_unexpected(1); // disengaged

void fun(expected<int, string>);
void fun(expected<int, int>);

fun(expected<int, string>::make_unexpected(1)); // unambiguous: a typeless make_unexpected would not do
```

Yet, we choose to propose a free function because we consider the above problems rare and a free function offers a very short notation in other cases:

```
expected<int, string> fun()
{
  expected<int, string> ei = make_unexpected(1);  // no ambiguity
  ei = make_unexpected(1);                // no ambiguity
  // ...
  return make_unexpected(1);              // no ambiguity
}
```

If the typeless function does not work for you, you can always use the following construct, although at the expense of invoking a (possibly elided) move constructor:

```
expected<int, expected<int, string>> eei =
    expected<int, string>{make_unexpected(1)};          // valued
expected<int, expected<int, string>> eej =
    expected<int, expected<int, string>>{make_unexpected(1)}; // unexpected

void fun(expected<int, string>);
void fun(expected<int,int>);

fun(expected<int, string>{make_unexpected(1)}); // unambiguous
```

5.3.5 Handling `initializer_list`

5.4 Observers

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

5.4.1 Explicit conversion to `bool`

The same rational described in [**?**] for `optional<T>` applies to `expected<E,T>` and so, the following example would combine initialization and checking for been valued in a condition.

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

5.4.2 Accessing the contained value

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

The same rational described in [**?**] for `optional<T>` applies to `expected<E,T>`.

5.4.3 Dereference operator

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

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

This pattern is used for all sort of pointers (smart or dumb), `optional` and 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 `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 overweighed by the benefit of an easy to grasp and 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 contains a value is an undefined behavior. This behavior offers maximum runtime performance.

5.4.4 Function value

In addition to the indirection operator, we propose the member function value as in [**?**] that returns a reference to the contained value if one exists or throws an exception otherwise:

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

  try {
    process_int(ei.value());
  }
  catch(bad_expected_access const&) {
    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 `expected<exception_ptr>`, it thows the exception stored on the `exception_ptr`.

This function can be implemented easily using the bool conversion and the dereference operator as a non member function

```
template <class E, class T>
contexpr T const& value(expected<E,T> const& e)
{
  if(e) return *e;
  else throw bad_expected_access(e.error());
}
template <class E, class T>
contexpr T & value(expected<E,T>& e)
{
  if(e) return *e;
  else throw bad_expected_access(e.error());
}
template <class T>
contexpr T const& value(expected<exception_ptr,T> const& e)
{
  if(e) return *e;
  else rethrow_exception(e.error());
}
template <class T>
contexpr T & value(expected<exception_ptr,T>& e)
{
  if(e) return *e;
  else rethrow_exception(e.error());
}
```

The advantage is the user could overload the function for other errors, as `any`, `variant<E1, ...., En>`. The liability is that free functions introduce a new name on the std namespace. Adding it to a specific namespace could solve this issue, but finding a good name is no so simple.

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

5.4.5 Getting the contained value on the context of a continuation

5.4.6 Accessing the contained error

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

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

5.4.7 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.

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

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

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

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

It is not really necessary as the line

```
return f.get_unexpected();
```

can be replaced by

```
return make_unexpected(f.error());
```

or even

```
return expected<errc, pair<int, int>>{unexpect, f.error()};
```

However, the function is provided for symmetry purpose. Implicit conversion from `unexpected<E>` to `expected<E,T>` and explicit conversion from `expected<E,T>` to `unexpected<E>`.

5.4.8 Function `has_exception`

[**?**] Expected class has a `hasException<E>` function that checks if the expected object has a stored exception that derived from `E`. This function has a sense only when the error parameter is a exception type erased as `std::exception_ptr` that contains any exception. This function is useful when the user don't needs to get more information than the type of the stored exception.

```
expected<exception_ptr,int> getIntOrZero(istream_range& r) {
    auto r = getInt();  // won't throw
    if (r.has_exception<EmptyStream>() {
        return 0;
    }
    return r;
}
```

5.4.9 Function `catch_exception`

When the user wants to retrieve the whole information on the stored exception the function `catch_exception` can be used instead.

```
expected<exception_ptr,int> getIntOrZero(istream_range& r)
{
  return getInt().
  catch_exception<NotANumber>([](auto& ex) // (1)
  {  // has complete access to the stored exception

  });   // try to recover
}
```

`catch_exception<E>` call to the function parameter if the expected instance has no value and the stored exception match the type.

5.4.10 Function `value_or`

This function template returns a value stored by the `expected` object if it is valued, and if not, it falls back to the default value specified in the second argument. This method for specifying default values on the fly rather than tying the default values to the type is based on the observation that different contexts or usages require different default values for the same type. For instance the default value for `int` can be `0` or `-1`. The callee might not know what value the caller considers special, so it returns the lack of the requested value explicitly. The caller may be better suited to make the choice what special value to use.

```
expected<exception_ptr,int> queryDB(std::string);
void setPieceCount(int);
void setMaxCount(int);

setPieceCount( queryDB("select piece_count from ...").value_or(0) );
setMaxCount( queryDB("select max_count from ...").value_or(numeric_limits<int>::max()) );
```

The decision to provide this function is controversial itself. As pointed out by Robert Ramey, the goal of the optional is to make the lack of the value explicit. Its syntax forces two control paths; therefore we will typically see an if-statement (or similar branching instruction) wherever expected is used. This is considered an improvement in correctness. On the other hand, using the default value appears to conflict with the above idea. One other argument against providing it is that in many cases you can use a ternary conditional operator instead:

```
auto&& cnt = queryDB("select piece_count from ...");
setPieceCount(cnt ? *cnt : 0);

auto&& max = queryDB("select max_count from ...");
setMaxCount(max ? std::move(*max) : numeric_limits<int>::max());
```

However, in case expected objects are returned by value and immediately consumed, the ternary operator syntax requires introducing an lvalue. This requires more typing and explicit move. This in turn makes the code less safe because a moved-from lvalue is still accessible and open for inadvertent misuse.

There are reasons to make it a free-standing function. (1) It can be implemented by using only the public interface of optional. (2) This function template could be equally well be applied to any type satisfying the requirements of `NullableProxy`. In this proposal, function `value_or` is defined as a member function. Making a premature generalization would risk standardizing a function with suboptimal performance/utility. While we know what detailed semantics (e.g., the return type) `value_or` should have for `expected`, we cannot claim to know the ideal semantics for any `NullableProxy`. Also, it is not clear to us if this convenience function is equally useful for pointers, as it is for optional objects. By making `value_or` a member function we leave the room for this name in namespace `std` for a possible future generalization.

The second argument in the function template's signature is not `T` but any type convertible to `T`:

```
template <class T, class E, class V>
  typename T expected<E,T>::value_or(V&& v) const&;
template <class T, class E, class V>
  typename T expected<E,T>::value_or(V&& v) &&;
```

This allows for a certain run-time optimization. In the following example:

```
expected<int, string> ex{"cat"};
string ans = ex.value_or("dog");

Because the expected object is valued, we do not need the fallback value and therefore to convert the string lit

It has been argued that the function should return by constant reference rather than value, which would avoid co

\begin{lstlisting}
void observe(const X& x);

expected<exception_ptr,X> ex { /* ... */ };
observe( ex.value_or(X{args}) );     // unnecessary copy
```

However, the benefit of the function `value_or` is only visible when the optional object is provided as a temporary (without the name); otherwise, a ternary operator is equally useful:

```
expected<exception_ptr,X> ex { /* ... */ };
observe(ox ? *ek : X{args});             // no copy
```

Also, returning by reference would be likely to render a dangling reference, in case the expected object is invalid, because the second argument is typically a temporary:

```
expected<exception_ptr,X> ex {};
auto&& x = ex.value_or(X{args});
cout << x;                                // x is dangling!
```

There is also one practical problem with returning a reference. The function takes two arguments by reference: the expected object and the default value. It can happen that one is deduced as lvalue reference and the other as rvalue reference. In such case we would not know what kind of reference to return. Returning lvalue reference might prevent move optimization; returning an rvalue reference might cause an unsafe move from lvalue. By returning by value we avoid these problems by requiring one unnecessary move in some cases.

We also do not want to return a constant lvalue reference because that would prevent a copy elision in cases where optional object is returned by value.

As for `std::experimental::optional<T>` the function `expected<E,T>::value_or<V>` could return type `decay_t<common_type_t` rather than `T`. This would avoid certain problems, such as loss of accuracy on arithmetic types:

```
// not proposed
expected<E,int> op = /* ... */;
long gl = /* ... */;

auto lossless = op.value_or(gl);    // lossless deduced as long rather than int
```

However, to be alined with `std::experimental::optional<T>` we do not propose it at this time.

Together with function `value`, `value_or` makes a set of similarly called functions for accessing the contained value that do not cause an undefined behavior when invoked on a invalid expected (at the expense of runtime overhead). They differ though, in the return type: one returns a value, the other a reference.

5.4.11 Relational operators

As `optional`, one of the design goals of `expected` is that objects of type `expected<E,T>` 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<E,T>` 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<E,T>` 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`.

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

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

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 know 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.

```
expected<int, unsigned> e0{0};
expected<int, unsigned> e1{1};
expected<int, unsigned> 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<E,T>`, this implies the existence and semantics of mixed comparison between `expected<E,T>` and `T`, as well as between `expected<E,T>` and `unexpected_type<E>`:

```
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<E,T>` and `unexpected_type<E>`, we still provide it for completness sake.

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

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

Given that `expected<E,T>` 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<E,T>` ; 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<nullopt_t,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<E,T>` and `unexpected_type<E>`, even though we see no practical use case for them:

```
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<E,T>::operato`

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<E,T>::operator<()` to be consistent with the choice taken for `std::optional`.

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

5.5 Modifiers

5.5.1 Reseting the value

Assigning the value of type `T` to `expected<E,T>` object results in doing two different things based on whether the expected object has a value or not. If expected object has a value, the contained value is assigned a new value. If expected object has an unexpected value, the destructor of the unexpected value is called and then it becomes valued using `T`'s copy/move constructor. This behavior is based on a silent assumption that `T`'s copy/move constructor is copying a value in a similar way to copy/move assignment. A similar logic applies to `expected<E,T>`'s copy/move assignment, although the situation here is more complicated because we have two valued/unexpected states to be considered. This means that `expected<E,T>`'s assignment does not work (does not compile) if `T` is not assignable:

```
expected<int,const int> ei = 1;  // ok
ei = 2;                          // error
ei = ei;                         // error
ei = make_unexpected(1);                    // ok
```

There is an option to reset the value of optional object without resorting to `T`'s assignment:

```
expected<int, const int> ej = 1;  // ok
ej.emplace(2);                    // ok
```

Function emplace disengages the optional object if it is engaged, and then just engages the object anew by copy-constructing the contained value. It is similar to assignment, except that it is guaranteed not to use T's assignment and provides only a basic exception safety guarantee. In contrast, assignment may provide a stronger guarantee if T's assignment does.

To sumarize, this proposal offers three ways of assigning a new contained value to an optional object:

```
optional<int> e;
e = make_optional(1);          // copy/move assignment
e = 1;                         // assignment from T
e.emplace(1);                  // emplacement
```

The first form of assignment is required to make optional a regular object, useable in STL. We need the second form in order to reflect the fact that optional<T> is a wrapper for T and hence it should behave as T as much as possible. Also, when optional<T> is viewed as T with one additional value, we want the values of T to be directly assignable to optional<T>. In addition, we need the second form to allow the interoperability with function std::tie as shown above. The third option is required to be able to reset an optional non-assignable T.

5.5.2 Tag `in_place`

This proposal makes use of the 'in-place' tag defined in [**?**]. 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<int,Big> eb{in_place, "1"}; // calls Big"1" in place (no moving)
expected<int,Big> ec{in_place};      // calls Big in place (no moving)
expected<int,Big> ed{};              // creates a unexpected expected
```

5.5.3 Tag `unexpect`

This proposal provides an 'unpect' 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, like calling `expected`'s default constructor and requesting `T`'s default construction:

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

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

5.5.4 Requirements

The `expected<std::exception_ptr,T>` specialization introduces some operations as `has_exception` and `catch_exception`. Should we name the classes differently? For example, `exception_or<T>` and `error_or<E,T>`.

5.5.5 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`. It is the particular operations on `expected<E,T>` that impose requirements on `T` and `E`: `expected<E,T>`'s move constructor requires that `T` and `E` are `MoveConstructible`, `expected<E,T>`'s copy constructor requires that `T` and `E` are `CopyConstructible`, and so on. This is because `expected<E,T>` is a wrapper for `T` or `E`: it should resemble `T` as much as possible. If `T` is `EqualityComparable` then (and only then) we expect `expected<E,T>` to be `EqualityComparable`.

5.5.6 Expected references

This proposal doesn't include expected references as [**?**] doesn't includes optional references neither.

5.5.7 Expected void

While it could seem weird to instantiate optional with `void`, it has more sense for expected as expected conveys in addition, as `future<T>`, an error code, `expected<E,void>`.

5.5.8 `NullableProxy`

As optional objects, the primary purpose of expected object is to check if they contain a value and if so, to provide access to this value. `expected<E,T>` could be seen also as a `NullableProxy`.

5.6 Literals

5.6.1 Making `expected` a literal type

We propose that `expected<E,T>` be a literal type for trivially destructible `T`'s and `E`'s.

```
constexpr expected<int, int> ei{5};
static_assert(ei, "");            // ok
static_assert(ei == ei, "");      // ok
int array[*ei];                   // ok: array of size 5
```

Making `expected<E,T>` a literal-type in general is impossible: the destructor cannot be trivial because it has to execute an operation that can be conceptually described as:

expected() if (valid()) destroy_contained_value(); else destroy_contained_error();

It is still possible to make the destructor trivial for `T`'s and `E`'s which provide a trivial destructor themselves, and we know an efficient implementation of such `expected<E,T>` with compile-time interface ? except for copy constructor and move constructor ? is possible. Therefore we propose that for trivially destructible `T`'s and `E`'s all `expected<E,T>`'s constructors, except for move and copy constructors, as well as observer functions are constexpr. The sketch of reference implementation is provided in [**?**].

We need to make a similar exception for `operator->` for types with overloaded `operator&`. The common pattern in the library is to use function `addressof` to avoid the surprise of overloaded `operator&`. However, we know of no way to implement constexpr version of function template addressof. The best approach we can take is to require that for normal types the non-overloaded (and constexpr) `operator&` is used to take the address of the contained value, and for the tricky types, implementations can use the normal (non-constexpr) `addressof`.

5.6.2 Moved from state

When a valued expected object is moved from (i.e., when it is the source object of move constructor or move assignment) its state does not change. When a valued object is moved from, we move the contained value, but leave the expected object valued. A moved-from contained value is still valid (although possibly not specified), so it is fine to consider such expected object valued. An alternative approach would be to destroy the contained value and make the moved-from optional object unvalid. However, we do not propose this for performance reasons.

In contexts, like returning by value, where you need to call the destructor the second after the move, it does not matter, but in cases where you request the move explicitly and intend to assign a new value in the next step, and if `T` does not provide an efficient move, the chosen approach saves an unnecessary destructor and constructor call:

```
expected<errc,array<Big, 1000>> eo = ... // array doesn't have efficient move
ep = std::move(eo);
eo = std::move(tmp);
```

The following is an even more compelling reason. In this proposal `expected<int,int>` is allowed to be implemented as a TriviallyCopyable type. Therefore, the copy constructor of type `std::array<expected<int,int>, 1000>` can be implemented using `memcpy`. With the additional requirement that expected's move constructor should not be trivial, we would be preventing the described optimization.

The fact that the moved-from expected is not invalid may look "uncomfortable" at first, but this is an invalid expectation. The requirements of library components expressed in 17.6.5.15 (moved-from state of library types) only require that moved-from objects are in a valid but unspecified state. We do not need to guarantee anything above this minimum.

5.7 Other

5.7.1 IO operations

The proposed interface for expected values does not contain IO operations: `operator<<` and `operator>>`. While we believe that they would be a useful addition to the interface of expected objects, we also observe that there are some technical obstacles in providing them, and we choose not to propose them at this time. Library components like optional, containers, pairs, tuples face the same issue. At present IO operations are not provided for these types. Our preference for `expected` is to provide an IO solution compatible with this for optional, containers, pairs and tuples, therefore at this point we refrain from proposing a solution for `expected` alone.

5.7.2 Function `make_expected`

5.8 Monad-like operations

[**?**] propose some improvements to `std::future<T>` that can be adapted to `expected<E,T>` naturally.

5.8.1 When ready

[**?**] provides a `future<T>.then()` function that accepts a continuation having the future object as parameter. This continuation function is called when the future becomes ready. Been expected always ready this function is less useful. The single role would be to adapt the result of the continuation to the expected.

5.8.2 When valued/unexpected

In addition the `.then()` function that accepts a continuation having the expected object as parameter, the proposal includes two separated functions, one `.mbind()` that applies when the expected object is valued and accepts a continuation having the underlying `value_type` as parameter. The other `.catch_error()` applies when the expected object is not valid and is used to try to recover from the error.

5.8.3 Continuation adaptors

An alternative to these specific functions could be to use the `.then()` function and have some adaptor `if_valued` and `if_unexpected` that do the adaptation.

```
f.then(if_valued([](T v) {...}));

f.then(if_unexpected([](E e) {...}));
```

TBoost.Expected [**?**] provides such a `.then` continuation adaptors, but this proposal doesn't include them.

5.8.4 `value_or_call`

As reported in [**?**] one convenience function has been suggested. Sometimes the default value is not given, and computing it takes some time. We only want to compute it, when we know the optional object is disengaged:

```
expected<exception_ptr,int> ei = /* ... */;

if (ei) {
  use(*ei);
}
else {
  int i = painfully_compute_default();
  use(i);
}
```

The solution to that situation would be another convenience function which rather than taking a default value takes a callable object that is capable of computing a default value if needed:

```
use( ei.value_or_call(&painfully_compute_default) );
// or
use( ei.value_or_call([&]{return painfully_compute_default();} );
```

This is quite close to the context of use of `catch_error`, but the function called has no parameter. As there is an alternative using generic lambdas this proposal doesn't propose neither this function.

```
ei.catch_error([](auto){ return painfully_compute_default();}).mbind(use);
```

5.8.5 When all ready

[**?**] includes `when_all()/when_any()` *//when_any_swaped()* functions to group futures on a new specific future that will be ready under different circumstances. As `expected<E,T>` is always ready, these functions have no sense. If provided the result and the behavior would be the same for both, just group all the expected.

5.8.6 When all valued

However, there is yet a need to apply a function when all the expected are valued. The free function `fmap()` takes a variadic continuation and a variadic number of expected parameters. The type and number of the parameters must be compatible with the continuation arguments.

```
 expected<exception_ptr,int> sumFirstAndSecond5(istream_range& r)
{
  return fmap(plus, getInt(r), getInt(r));
}
```

An alternative to this specific function if `when_all` is provided, could be to use the `.then()` function and have some adaptor `if_all_valued` that do the adaptation.

```
expected<exception_ptr,int> sumFirstAndSecond5(istream_range& r)
{
  return when_all(getInt(r), getInt(r)).then(if_all_valued([](int i, int j) {...}));
}
```

5.8.7 When any valued

The authors don't have a concrete use case for a function that would be applied if any of the expected has a value other than looking for non-determinism.

5.8.8 `expected<E, expected<E,T>>`

5.8.9 Function `unwrap`

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

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

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

6 Related types

6.1 Variant

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

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

6.2 Optional

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

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

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

	boost::variant<unexpected<E>, T>	**expected<E,T>**
never-empty warranty	yes	yes
accepts is_same<T,E>	no	yes
swap	yes	yes
factories	no	make_expected / make_unexpected
hash	yes	yes
value_type	no	yes
default constructor	yes (if T is default constructible)	yes (if T is default constructible)
observers	boost::get<T> and boost::get<E>	pointer-like / value / error / value_or / value_or_throw
continuations	apply_visitor	then/mbind/catch_error

Table 2: Comparison between variant and expected.

We can convert an `optional<T>` to an `expected<exception_ptr,T>` without knowledge of the root cause.

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

6.3 Promise and Future

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

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

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

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

where

```
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

	optional	expected	promise/future
specific null value	yes	no	no
relational operators	yes	yes	no
swap	yes	yes	yes
factories	make_optional / nullopt	make_expected / make_unexpected	make_ready_future / (make_exceptional, see [**?**])
hash	yes	yes	yes
value_type	yes	yes	no / (yes, see [**?**]).
default constructor	yes	yes (if T is default constructible)	yes
allocators	no	no	yes
emplace	yes	yes	no
bool conversion	yes	yes	no
state	bool()	bool()	valid / ready / (has_value, see [**?**])
observers	pointer-like / value / value_or	pointer-like / value / error / value_or / value_or_throw	get / (get_exception_ptr, see [**?**])
visitation	no	then/mbind/catch_error	then / (next/recover see [**?**])
grouping	n/a	n/a	when_all / when_any
apply	no	mbind	no

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

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

6.4 Expected monad

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

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

6.5 Comparison between optional, expected and future

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

7 Open questions

7.1 Allocator support

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

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

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

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

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

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

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

the authors consider that this is not really needed and that this parameter should not really be part of the type. In addition `OPTIONAL`

The user can use `value_or_throw()` as

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

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

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

and use it as

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

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

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

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

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

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

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

- a T, or
- an error code, or
- a `exception_ptr`

Again there is no major difficulty to implement it, but instead of having one variation point we have two, that is, is there a value, and if not, if is there an exception_ptr. While this would need only an extra test on the exceptional case, the authors think that it is not worth doing it as all the copy/move/swap operations would be less efficient.

8 Proposed Wording

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

Insert a new section.

X.Y Unexpected objects [unexpected]

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

This subclause describes class template `unexpected_type` that wraps objects intended as unexpected. This wrapped unexpected object is used to be implicitly convertible to other object.

X.Y.2 Header <experimental/unexpected> synopsis [unexpected.synop]

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

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

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

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

```
template <class E=std::exception_ptr>
class unexpected_type {
public:
    unexpected_type() = delete;
    constexpr explicit unexpected_type(E const&);
    constexpr explicit unexpected_type(E&&);
    constexpr E const& value() const;
};

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

Effects:

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

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

Effects:

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

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

Returns:

A const reference to the stored error.

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

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

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

Effects:

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

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

Effects:

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

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

Effects:

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

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

Returns:

A const reference to the stored `exception_ptr`.

X.Y.5 Factories [unexpected.factories]

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

Returns:

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

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

Returns:

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

Insert a new section.

X.Y Expected objects [expected]

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

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

X.Y.7 Header <experimental/expected> synopsis [expected.synop]

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

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

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

  // X.Y.14, class expected_default_constructed
  class expected_default_constructed;

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

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

  // X.Y.17, Comparison with unexpected_type<E>
  template <class T, class E> constexpr bool operator==(const expected<E,T>&, const unexpected<E>&);
  template <class T, class E> constexpr bool operator==(const unexpected<E>&, const expected<E,T>&);
  template <class T, class E> constexpr bool operator!=(const expected<E,T>&, const unexpected<E>&);
```

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


  // X.Y.18, Specialized algorithms
  template <class T>
    void swap(expected<E,T>&, expected<E,T>&) noexcept(see below);

  // X.Y.19, Factories
  template <class T> constexpr expected<exception_ptr, decay_t<T>> make_expected(T&& v);
  template <> expected<exception_ptr, void> make_expected();
  template <class E> expected<E,void> make_expected();

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

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

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

  // X.Y.20, hash support
  template <class T> struct hash;
  template <class T> struct hash<expected<E,T>>;
}}}
```

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

X.Y.8 Definitions [expected.defs]

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

X.Y.9 expected for object types [expected.object]

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

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

     // X.Y.9.1, constructors
    constexpr expected() noexcept(see below);
    expected(const expected&);
    expected(expected&&) noexcept(see below);
```

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

constexpr expected(unexpected_type<E> const&);
template <class Err>
constexpr expected(unexpected_type<Err> const&);

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

// X.Y.9.3, assignment
expected& operator=(const expected&);
expected& operator=(expected&&) noexcept(see below);

template <class U> expected& operator=(U&&);

expected& operator=(const unexpected_type<E>&);
expected& operator=(unexpected_type<E>&&) noexcept(see below);

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

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

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

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

constexpr explicit operator bool() const noexcept;

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

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

template <class G> constexpr T value_or_throw() const&;
template <class G> T value_or_throw() &&;

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

// X.Y.9.6, factories

template <typename Ex, typename F>
```

```
    expected<E,T> catch_exception(F&& f);

    template <typename F>
      auto mbind(F&& func) const -> expected<E, decltype(func(val))>;
    template <typename F>
      expected<E,T> catch_error(F&& f);
    template <typename F>
      auto then(F&& func) const -> expected<E, decltype(func(*this))>;

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

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

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

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

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

```
constexpr expected<E,T>::expected() noexcept(see below);
```

Effects:

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

Postconditions:

`bool(*this)`.

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

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

Postconditions:

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

Throws:

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

Remarks:

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

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

Effects:

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

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

Postconditions:

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

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

Postconditions:

`bool(*this)`.

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

Postconditions:

`bool(*this)`.

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

Postconditions:

`bool(*this)`.

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

Postconditions:

`bool(*this)`.

Throws:

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

Remarks:

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

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

Remarks:

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

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

Effects:

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

Postconditions:

`! *this`.

Throws:

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

Remarks:

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

Remarks:

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

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

Effects:

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

Postconditions:

`! *this`.

Throws:

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

Remarks:

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

Remarks:

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

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

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

Effects:

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

Remarks:

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

X.Y.9.3 Assignment [expected.object.assign]

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

Effects:

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

Returns:

`*this`.

Postconditions:

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

Exception Safety:

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

Remarks:

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

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

Effects:

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

Returns:

`*this`.

Postconditions:

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

Remarks:

The expression inside `noexcept` is equivalent to:
`is_nothrow_move_assignable<T>::value &&`
`is_nothrow_move_constructible<T>::value &&`
`is_nothrow_move_assignable<E>::value &&`
`is_nothrow_move_constructible<E>::value`.

Exception Safety:

If any exception is thrown, the values of `bool(*this)` and `bool(rhs)` remain unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `rhs.val` is determined by exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `val` and `rhs.val` is determined by exception safety guarantee of `T`'s move assignment. If an exception is thrown during the call to `E`'s move constructor, the state of `rhs.err` is determined by exception safety guarantee of `E`'s move constructor. If an exception is thrown during the call to `E`'s move assignment, the state of `err` and `rhs.err` is determined by exception safety guarantee of `E`'s move assignment.

Remarks:

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

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

Effects:

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

Returns:

`*this`.

Postconditions:

`bool(*this)`.

Exception Safety:

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

Remarks:

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

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

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

Effects:

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

Returns:

`*this`.

Postconditions:

`! *this`.

Exception Safety:

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

Remarks:

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

```
template <class... Args>
void expected<E,T>::emplace(Args&&... args);
```

Effects:

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

Postconditions:

`bool(*this)`.

Exception Safety:

If an exception is thrown during the call to `T`'s constructor, `*this` is disengaged, and the previous `val` (if any) has been destroyed.

Throws:

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

Remarks:

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

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

Effects:

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

Postconditions:

`bool(*this)`.

Exception Safety:

If an exception is thrown during the call to `T`'s constructor, `! *this` , and the previous `val` (if any) has been destroyed.

Throws:

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

Remarks:

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

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

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

Effects:

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

Exception Safety:

TODO: This must be reworded. If any exception is thrown, values of `has_value` and `rhs.has_value` remain unchanged. If an exception is thrown during the call to function swap the state of `val` and `rhs.val` is determined by the exception safety guarantee of swap for lvalues of `T`. If an exception is thrown during the call to `T`'s move constructor, the state of `val` and `rhs.val` is determined by the exception safety guarantee of `T`'s move constructor.

Throws:

Any exceptions that the expressions in the Effects clause throw.

Remarks:

The expression inside `noexcept` is equivalent to:

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

Remarks:

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

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

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

Requires:

`bool(*this)`.

Returns:

`&val`.

Remarks:

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

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

Requires:

`bool(*this)`.

Returns:

`val`.

Remarks:

The first function shall be a `constexpr` function.

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

Returns:

`has_value`.

Remarks:

This function shall be a `constexpr` function.

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

Returns:

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

Throws:

`bad_expected_access(err)` if `!*this`.

Remarks:

The first function shall be a `constexpr` function.

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

Returns:

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

Throws:

`bad_expected_access(err)` if `!*this`.

Remarks:

The first function shall be a `constexpr` function.

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

Requires:

`!*this`.

Returns:

`err`.

Remarks:

The first function shall be a `constexpr` function.

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

Requires:

`!*this`.

Returns:

`move(err)`.

Remarks:

The first function shall be a `constexpr` function.

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

Returns:

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

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

Requires:

`!*this`.

Returns:

`make_unexpected(err)`.

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

Returns:

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

Exception Safety:

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

Throws:

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

Remarks:

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

Remarks:

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

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

Returns:

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

Exception Safety:

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

Throws:

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

Remarks:

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

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

Returns:

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

Exception Safety:

If `has_value` and exception is thrown during the call to `T`'s constructor, the value of `has_value` 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.

Throws:

If `! *this` then `G(error())`.
Any exception thrown by the selected constructor of `T`.

Remarks:

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

Remarks:

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

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

Returns:

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

Exception Safety:

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

Throws:

If `! *this` then `G(error())`.
Any exception thrown by the selected constructor of `T`.

Remarks:

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

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

Returns:

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

Throws:

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

Remarks:

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

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

Returns:

`*this`.

Throws:

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

Remarks:

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

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

Returns:

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

Throws:

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

Remarks:

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

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

Returns:

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

Throws:

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

Remarks:

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

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

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

Effects:

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

Returns:

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

```
template <class Ex,class F>
auto expected<E,T>::then(F&& func) const -> unwrap_nested_expected_t<expected<E, decltype(func(val))>>;
```

Returns:

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

```
template <class Ex,class F>
auto expected<E,T>::mbind(F&& func) const -> unwrap_nested_expected_t<expected<E, decltype(func(val))>>;
```

Returns:

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

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

Returns:

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

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

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

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

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

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

   // ??, constructors
  constexpr expected() noexcept;
  expected(const expected&);
  expected(expected&&) noexcept(see below);
  constexpr explicit expected(in_place_t);

  constexpr expected(unexpected_type<E> const&);
  template <class Err>
  constexpr expected(unexpected_type<Err> const&);

  // ??, destructor
  ~expected();

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

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

```
    // ??, observers
    constexpr explicit operator bool() const noexcept;
    void value() const;
    constexpr E const& error() const&;
    constexpr E& error() &;
    constexpr E&& error() &&;
    constexpr unexpected<E>  get_unexpected() const;

    template <typename Ex>
    bool has_exception() const;

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

    // ??, factories


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

    template <typename F>
      auto mbind(F&& func) const -> expected<E, decltype(func())>;
    template <typename F>
      expected<void,E> catch_error(F&& f);
    template <typename F>
      auto then(F&& func) const -> expected<E, decltype(func(*this))>;

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

TODO: Describe the functions.

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

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

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

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

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

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

Effects:

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

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

Returns:

The stored error..

Remarks:

The first function shall be a `constexpr` function.

X.Y.14 Class expected_default_constructed [expected.expected_default_constructed]

```
namespace std {
  template <class E>
  class expected_default_constructed : public logic_error {
  public:
    explicit expected_default_constructed();
  };
}
```

The class `expected_default_constructed` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the `expected<exception_ptr,T>::error()`/`expected<exception_ptr,T>::get_unexpecte` value of a unexpected expected object that has no exception stored.

TODO: Describe the functions.

X.Y.15 Expected Relational operators [expected.relational_op]

TODO: Describe the functions.

X.Y.16 Comparison with `T` [expected.comparison_T]

TODO: Describe the functions.

X.Y.17 Comparison with `unexpected<E>` [expected.comparison_unexpected_E]

TODO: Describe the functions.

X.Y.18 Specialized algorithms [expected.specalg]

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

Effects:

calls `x.swap(y)`.

X.Y.19 Expected Factories [expected.factories]

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

Returns:

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

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

Returns:

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

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

Returns:

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

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

Returns:

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

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

Returns:

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

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

Equivalent to:

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

X.Y.20 Hash support [expected.hash]

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

Requires:

TODO: This must be reworded

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

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

Requires:

9 Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++14. An almost full reference implementation of this proposal can be found at TBoost.Expected [**?**].

10 Acknowledgement

We are very grateful to Andrei Alexandrescu for his talk, which was the origin of this work. We thanks also to every one that has contributed to the Haskell either monad, as either's interface was a source of inspiration. Thanks to Fernando Cacciola, Andrzej KrzemieÃĚâĂđski and every one that has contributed to the wording and the rationale of N3793 [**?**].

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