Document number:	D0323R1
Date:	2016-05-30
Revises:	N4109/N4015
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

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

Abstract

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

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

Table of Contents

History

Revision 3 - Revision of [P0323R0] after with feedback from Oulu

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

- Split the proposal on a simple `expected` class and a generic monadic interface.
- As `variant`, `expected` requires some properties in order to ensure the never-empty warranties. As the error type should be no throw movable, we are always sure to be able to ensure the never-empty warranties.
- Removed future work section

Revision 2 - Revision of N4109 after discussion on the ML

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

Not done yet

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

Revision 1 - Revision of N4015 after Rapperswil feedback:

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

Introduction

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

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

Motivation

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

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

The first and the third characteristic seem to be quite contradictory and deserve further explanation. The former points out that errors not handled should appear clearly in the code. The latter tells us that the error handling mustn't interfere with the code reading, meaning that it clearly shows the normal execution flow. A comparison between the exception and return codes is given in the next table.

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

Expected class

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

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

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

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

Use cases

Safe division

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

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

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

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

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

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

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

```
function i + j/k is:
double f1(double i, double j, double k)
{
    return i + safe_divide(j,k);
}
```

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

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

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

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

Error retrieval and correction

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

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

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

`exception_ptr` and the `error_condition` types.

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

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

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

Impact on the standard

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

Design rationale

The same rationale described in N3672 for `optional<T>` applies to `expected<T,E>` and `expected<T, nullopt_t>` should behave almost as `optional<T>` with some exceptions. That is, we see `expected<T,E>` as `optional<T>` for which all the values of `E` collapse into a single value `nullopt` . In the following sections we present the specificities of the rationale in N3672 applied to `expected<T,E>` .

Conceptual model of expected

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

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

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

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

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

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

Initialization of `expected<T,E>`

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

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

```
string s"STR";

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

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

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

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

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

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

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

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

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

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

Almost never-empty guaranty

TODO: revise this section as it cannot always be ensured

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

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

In order to ensure this property the types 'T' and `E` must satisfy some requirements as described in P0110R0. Given the nature of the parameter `E`, that is, to transport an error, it is expected that `is_nothrow_copy_constructible<E>` or at least `is_nothrow_move_constructible<E>`.

The default constructor

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

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

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

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

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

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<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 N3527 valid for `optional<T>`, however propose that `expected<T,E>` default constructor should behave as constructed with `T()` if `T` is default constructible.

Conversion from `T`

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

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

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

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

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

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

Conversion from `E`

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

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

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

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

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

Should we support the `exp2 = {}`?

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

TODO: What is the meaning of exp2 = {}, now that expected defaults to T{}?

Observers

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

Explicit conversion to bool

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

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

`has_value` following P0032

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

Accessing the contained value

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

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

Dereference operator

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

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

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

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

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

Function value

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

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

The exception thrown depends on the expected error type. By default it throws `bad_expected_access<E>` (derived from `std::logic_error`) which will contain the stored error. In the case of `expected<T, exception_ptr>`, it throws the exception stored in the `exception_ptr`. An approach enabling customization of this behavior is presented in the section [Customizing the exception thrown]. `bad_expected_access<E>` and `bad_optional_access` could inherit both from a `bad_access` exception derived from `logic_error`, but this is not proposed yet.

Accessing the contained error

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

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