Document number:	P1051R0
Date:	2018-05-03
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

`std::experimental::expected` LWG design issues

Abstract

LWG raised 3 design issues concerning the proposed `std::experimental::expected` [P0323R5].

- Why `unexpected<void>` is not valid?
- Why there is no `expected<T,E>:emplace` for `unexpected<E>` ?
- The in-place construction of success (`in_place_t`) and failure (`unexpect_t`) naming is not symmetric. In addition `unexpect_t` doesn't convey any idea of emplacement.

This paper is a response to these issues.

Table of Contents

Introduction

Motivation

LWG raised 3 design issues concerning the proposed `std::experimental::expected` [P0323R5].

- Why `unexpected<void>` is not valid?

There is no major reason to don't allow it. We can see `unexpected<void>` as a degenerate case when the error doesn't conveys information as it is the case for `nullopt_t`.

- Why there is no `expected<T,E>:emplace` for `unexpected<E>`?

There is no major reason to don't be uniform. Emplacing failure cases could be as usual as emplacing success cases.

- The in-place construction of success (`in_place_t`) and failure (`unexpect_t`) naming is not symmetric. In addition `unexpect_t` doesn't convey any idea of emplacement.

The author agrees. Originally, the design was to use a `expect_t` tag for success and another `unexpect_t` for failure, but the optional design imported the `in_place_t` tag and lost the `expect_t` tag.

The purpose of the papers is to analyze the impact of this LWG questions.

For emplacing, the paper analyzes two alternatives, one based on the fact that there are only two alternatives as it is the case for `std::promise/std::future` and the fact that the two alternatives are open, as for `std::variant`, that is the error type is not a `exception_ptr`.

Proposal

Make `unexpected<void>` specialization valid

When the `E` in `expected<T,E>` is `void` we cannot retrieve it and so the expression

`unexpected(rhs.error())`

has no sense, is ill formed.

In order to make the wording simpler, it is better to raise the level of abstraction and work directly with `unexpected<E>` and have a way to retrieve it

`rhs.get_unexpected()`

Note that the original proposal had already a function to get it, with just this name, but the committee requested the removal of this function.

This paper raises the need for this function again.

Emplacement based on `promise` / `future` similarity

Given that we have only two choices, as it is the case for `std::future`, the interface could take advantage of this fact and define two specific `success_t` / `failure_t` tags.

```
    template <class... Args>
        constexpr explicit expected(in_place, success_t, Args&&...);
    template <class S, class U, class... Args>
        constexpr explicit expected(in_place, success_t, initializer_list<U>, Args&&
    template <class... Args>
        constexpr explicit expected(in_place, failure_t, Args&&...);
    template <class S, class U, class... Args>
        constexpr explicit expected(in_place, failure_t, initializer_list<U>, Args&&

    template <class... Args>
        T& emplace(success_t, Args&&...);
    template <class U, class... Args>
        T& emplace(success_t, initializer_list<U>, Args&&...);
    template <class... Args>
        unexpected<E>& emplace(failure_t, Args&&...);
    template <class U, class... Args>
        unexpected<E>& emplace(failure_t, initializer_list<U>, Args&&...);
```

If we follow the `promise::set_value` / `promise::set_exception` interface design, we could as well have more specific functions for emplace.

```
    template <class... Args>
        T& emplace_success(Args&&...);
    template <class U, class... Args>
        T& emplace_success(initializer_list<U>, Args&&...);
    template <class... Args>
        unexpected<E>& emplace_failure(Args&&...);
    template <class U, class... Args>
        unexpected<E>& emplace_failure(initializer_list<U>, Args&&...);
```

For construction, we should strive to factories if we want a more specific name as we have `make_ready_future` / `make_exceptional_future`. The problem is the `expected<T,E>` error parameter that is not fixed as it is for `std::future<T>`.

```
template <class T, class E, class... Args>
expected<T, E> make_success_expected(Args&&...);
template <class T, class E, class... Args>
expected<T, E> make_failure_expected(Args&&...);
```

```
    stdex::expected<T,E> e1 {std::in_place, stdex::success_t, a, b, c};
    stdex::expected<T,E> e2 {std::in_place, stdex::failure_t, a, b, c};

    e1.emplace<stdex::faulre_t>(d,e);
    e1.emplace<unexpected<E>>(d,e);
    e1.emplace<T>(d,e);

    auto e3 = make_success_expected<T,E>(a, b, c);
    auto e4 = make_failure_expected<T,E>(a, b, c);

    e3.emplace_failure(d,e);
    e4.emplace_success(d,e);
```

Emplacement based on `variant<T, unexpected<E>` similarity

The `expected` design was based originally on the `std::experimental::optional` design as there is a specific success alternative and `std::variant` was not yet there (2014). However `expected<T,E>` is also close to `std::variant` by he fact that we have more than one alternative.

An alternative to the current in-place construction interface is to adapt the one for `std::variant` using the `in_place_type_t<T>` tag, for the success and the failure alternatives, and follows the `variant::emplace` interface that requires the type to be emplaced, the success or the failure.

```
    template <class S, class... Args>
        constexpr explicit expected(in_place_type_t<S>, Args&&...);
    template <class S, class U, class... Args>
        constexpr explicit expected(in_place_type_t<S>, initializer_list<U>, Args&&.
    template <size_t I, class... Args>
        constexpr explicit expected(in_place_index_t<I>, Args&&...);
    template <size_t I, class U, class... Args>
        constexpr explicit expected(in_place_index_t<I>, initializer_list<U>, Args&&

     template <class S, class... Args>
        S& emplace(Args&&...);
    template <class S, class U, class... Args>
        S& emplace(initializer_list<U>, Args&&...);
    template <size_t I, class... Args>
        see bellow& emplace(Args&&...);
    template <size_t I, class U, class... Args>
        see bellow& emplace(initializer_list<U>, Args&&...);
```

We could have two named index

```
constexpr size_t success_index = 0;
constexpr size_t failure_index = 1;
```

and two

```
using in_place_success_t = in_place_index_t<success_index>;
using in_place_failure_t = in_place_index_t<failure_index>;

constexpr in_place_success_t in_place_success;
constexpr in_place_failure_t in_place_failure;
```

that could make the use with indexes more friendly

```
stdex::expected<T,E> e1 {std::in_place_index_t<0>, a, b, c};
stdex::expected<T,E> e2 {stdex::in_place_index_t<stdex::success_index>, a, b, c}
stdex::expected<T,E> e3 {stdex::in_place_success, a, b, c};
stdex::expected<T,E> e4 {std::in_place_type_t<T>, a, b, c};
stdex::expected<T,E> e5 {stdex::in_place_failure, d, e};

e1.emplace<1>(d,e);
e2.emplace<stdex::failure_index>(d,e);
e3.emplace<stdex::unexpected<E>>(d,e);
e4.emplace<T>(a, b, c);
```

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

Emplacement based on `optional` interface

This design would give a priority to the success alternative, making the failure alternative asymmetric and less friendly.

Emplacement based on `promise/future` interface

Given that we have only two choices, as it is the case for `std::future`, the interface could take advantage of this fact and define two specific `success_t` / `failure_t` tags.

Emplacement based on `variant` interface

The `expected` design was based originally on the `std::experimental::optional` design, as the `std::variant` was not yet there (2014). However `expected<T,E>` is closer to `std::variant` than to `optional`.

An alternative to the current in-place construction interface is to adapt the one for `std::variant` using the `in_place_type_t<T>` tag, for the success and the failure alternatives, and follows the `variant::emplace` interface that requires the type to be emplaced, the success or the failure.

Open points

The authors would like to have an answer to the following points:

Do we want `unexpected<void>`?

Do we want the alternative emplace interface based on `std::variant` using `in_place_type_t<T>`?

Do we want the alternative emplace interface based on `std::variant` using `in_place_index_t<I>`?

Do we want names for the success/failure indexes?

Proposed Wording

The proposed changes are expressed as edits to N4617 the Working Draft - C++ Extensions for Library Fundamentals V3.

The impact of allowing `unexpected<void>` and `expected<T, void>` is quite viral at the wording level, as it has an impact on almost all the functions that have to work with `E`.

The impact of emplace functions is restricted to those functions and so is minor compared with the previous one.

Next follows the interface changes to have an idea of the impact. Wording to be completed after we decide the desired approach.

🔲.🔲.2 Header `<experimental/expected>` synopsis [*expected.synop*] {#expected.synop}

Update the Synopsis

```
 // ?.?.4, class template expected
template <class T, class E>
class expected;
 // ?.?.8
constexpr size_t success_index = 0;
constexpr size_t failure_index = 1;
using in_place_success_t = in_place_index_t<success_index>;
using in_place_failure_t = in_place_index_t<failure_index>;

constexpr in_place_success_t in_place_success;
constexpr in_place_failure_t in_place_failure;
```

?.?.4 Class template expected [*expected.expected*] {#expected.expected}

Update synopsis

```
template <class T, class E>
class expected
{
...
    template <class S, class... Args>
        constexpr explicit expected(in_place_type_t<S>, Args&&...);
    template <class S, class U, class... Args>
        constexpr explicit expected(in_place_type_t<S>, initializer_list<U>, Args&&.
    template <size_t I, class... Args>
        constexpr explicit expected(in_place_index_t<I>, Args&&...);
    template <size_t I, class U, class... Args>
        constexpr explicit expected(in_place_index_t<I>, initializer_list<U>, Args&&

     template <class S, class... Args>
        S& emplace(Args&&...);
    template <class S, class U, class... Args>
        S& emplace(initializer_list<U>, Args&&...);
    template <size_t I, class... Args>
        see bellow& emplace(Args&&...);
    template <size_t I, class U, class... Args>
        see bellow& emplace(initializer_list<U>, Args&&...);

...
};
```

?.?.4.1 Constructors [*expected.object.ctor*]

{#expected.object.ctor}

Replace

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

by

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

and

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

by

```
template <class... Args>
    constexpr explicit expected(in_place_type_t<unexpected<E>>, Args&&... args);
template <class U, class... Args>
    constexpr explicit expected(in_place_type_t<unexpected<E>>, initializer_list<U>
```

Add

```
template <class... Args>
    constexpr explicit expected(in_place_index_t<0>, Args&&... args);
template <class U, class... Args>
    constexpr explicit expected(in_place_index_t<0>, initializer_list<U> il, Args&&.
```

and

```
template <class... Args>
    constexpr explicit expected(in_place_index_t<1>, Args&&... args);
template <class U, class... Args>
    constexpr explicit expected(in_place_index_t<1>, initializer_list<U> il, Args&&
```

ﷺ.ﷺ.4.3 Assignment [*expected.object.assign*] {#expected.object.assign}

Replace

```
void expected<void,E>::emplace();
```

by

```
template <>
void expected<void, E>::emplace<void>();
template <>
void expected<T, void>::emplace<unexpected<void>>();
template <>
void expected<void, E>::emplace<success_index>();
template <>
void expected<T, void>::emplace<failure_index>();
```

and

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

by

```
 template <class S, class... Args>
    S& emplace(Args&&...);
template <class S, class U, class... Args>
    S& emplace(initializer_list<U>, Args&&...);
template <size_t I, class... Args>
    see bellow& emplace(Args&&...);
template <size_t I, class U, class... Args>
    see bellow& emplace(initializer_list<U>, Args&&...);
```

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++17.

An almost full reference implementation of the different approaches of this proposal can be found at Expected2Impl.

Acknowledgements

We are very grateful to the LWG, and in particular to Geoff Romer, for raising these design issues that are the "raison d`être" of this paper.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- N4617 N4617 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 DTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf

- Expected2Impl Vicente J. Botet Escriba. Expected, 2018.

  https://github.com/viboes/std-make/tree/master/include/experimental/fundamental/v3/expected2

History

Revision 0 - Created after Jacksonville LWG P0323R5's feedback