Document number:	P0786R0
Date:	2018-02-12
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

ValuedOrError and *ValueOrNone* types

Abstract

There are types that contain a success value or a failure value.

In the same way we have *Nullable* types that have a single not-a-value we have types that can contain a single instance of value-type and a mean to retrieve it using the `deref` function are named here as *ValueOrNone*.

Types that are possibly valued and have a single error are named in this paper *ValueOrError*. They provide the `error` function. These types have something in common with *Nullable* and is the ability to know if they have a value or not via the `has_value` function.

`std::optional` is a *ValueOrNone* type. The proposed `std::experimental::expected` P0323R4 is a *ValuedOrError* type.

Table of Contents

Introduction

This paper proposes the concept of *ValueOrError* that represents a type that can contain a success value or a failure value that can be used as the result of a function to return the value computed by the function or the reason of the failure of this computation.

ValueOrError contains the interface needed to customize the types that can work with the proposed `operator try`. This makes the error propagation on functions returning this kind of types much more simpler.

The paper proposes also some error handling utilities that help while the user wants to recover from error as `resolve`, `value_or`, `value_or_throw`, `error_or` and `check_error`.

Some *ValueOrError* types contain success and/or failure types that wrap a value or an error. However, the user wants to see the wrapped value and error types instead of the wrapping success and failure types. These types allow to unwrap the type to get the underlying wrapped value.

When the type is *TypeConstructible* and *ValueOrError*, the type can be seen as a *Functor*, an *ApplicativeFunctor*, a *Monad* or a *MonadError*.

ValueOrError as a *SumType* can provide the `visit` function. However we cannot specialize the variant-like traits, nor the `get<I>` functions. Nevertheless we could specialize the *SumType* traits, once we have a proposal.

BEFORE	AFTER
Customizations	

`o.has_value()`
`e.has_value()`
 	
`*o`
`*e`
 	
`nullopt`
`e.error()`
 	
`nullopt`
`unexpected(e.error())`
	`value_or_error::has_value(o)`
	`value_or_error::has_value(e)`
	
	`value_or_error::deref(o)`
	`value_or_error::deref(e)`
	
	`value_or_error::error(o)`
	`value_or_error::error(e)`
	
	`value_or_error::failure_value(o)`
	`value_or_error::failure_value(e)`

Functor/Monad	

`(o) ? f(*o) : nullopt`
`(o) ? f(*o) : unexpected(e.error)`
`(o) ? g(*o) : nullopt`
`(o) ? g(*o) : unexpected(e.error)`
	`value_or_error::transform(o, f)`
	`value_or_error::transform(e, f)`
	`value_or_error::bind(e, g)`
	`value_or_error::bind(e, g)`

Error Handling	

`o.value_or(v)`
`e.value_or(v)`
 	
`(!e) ? e.error() : err`
 	
`(e) ? false : e.error() == err`
	`value_or_error::value_or(o, v)`
	`value_or_error::value_or(e, v)`
	
	`value_or_error::error_or(e, err)`
	
	`value_or_error::check_error(e, err)`

Sum types	
`auto r = (e) ? f(*e) : v(unexpected(e.error));`	`auto r = value_or_error::visit(e, v);`

Motivation and Scope

Propagating failure using `optional` and `expected` as return values

```
optional<expr_plus<int>> fo(...)
{
    auto o1 = expr1(...);
    if ( ! o1.has_value() )
        return nullopt;
    auto& v1 = *o1;
    auto o2 = expr2(...);
    if ( ! o2.has_value() )
        return nullopt;
    auto& v2 = *o2;
    return expr_plus<int>(v1, v2);
}
```

```
expected<expr_plus<int>, error_code> fe(...)
{
    auto e1 = expr1(...);
    if ( ! e1.has_value() )
        return unexpected(e1.error());
    auto& v1 = *e1;
    auto e2 = expr2(...);
    if ( ! e2.has_value() )
        return unexpected(e1.error());
    auto& v2 = *e2;
    return expr_plus<int>(v1, v2);
}
```

ValueOrError types

What `optional` and `expected` have in common?

Both types have a way to states if the operation that produced them succeeded or failed, they allow to get the success value and to get the failure value.

`optional<T>` can be seen as the sum type of the failure type `nullopt_t` and the success type `T`. `expected<T,E>` can be seen as the sum type of the failure type `unexpected<E>` and the success type `T`.

In the case of `expected`, the failure type wraps the error type.

We propose a concept *ValueOrError* that allows to customize the 4 functions and provide access via

- `value_or_error::succeeded` / `value_or_error::failed`
- `value_or_error::success_value`
- `value_or_error::failure_value`

`value_or_error::failed` must be the negation of `value_or_error::succeeded`.

Error propagation with *ValueOrError* types

Once we have the mapping

```
optional<expr_plus<int>> fo(...)
{
    using namespace value_or_error:
    auto e1 = expr1(...);
    if ( failed(e1) )
        return failure_value(e1);
    auto& v1 = success_value(e1);
    auto e2 = expr2(...);
    if ( failed(e2) )
        return failure_value(e2);
    auto& v2 = value_or_error::success_value(e2);
    return expr_plus<int>(v1, v2);
}
```

```
expected<expr_plus<int>, error_code> fe(...)
{
    using namespace value_or_error:
    auto e1 = expr1(...);
    if ( failed(e1) )
        return failure_value(e1);
    auto& v1 = success_value(e1);
    auto e2 = expr2(...);
    if ( failed(e2) )
        return failure_value(e2);
    auto& v2 = success_value(e2);
    return expr_plus<int>(v1, v2);
}
```

Now that both have the same interface we can make a generic function that works on *ValueOrError* types

```
template <class TC>
invoke_t<TC, expr_plus<int>> f(...)
{
    using namespace value_or_error:
    auto e1 = expr1(...);
    if ( failed(e1) )
        return failure_value(e1);
    auto& v1 = success_value(e1);
    auto e2 = expr2(...);
    if ( failed(e2) )
        return failure_value(e2);
    auto& v2 = success_value(e2);
    return expr_plus<int>(v1, v2);
}
```

The previous function requires that the *ValueOrError* in constructible from `expr_plus<int>`. This is the case for `optional` and `expected`. However having a *ValueOrError* `result<T,E>` that is built from `success<T>` and `failure<E>` wouldn't work. We will need a to customize a factory function as `make`

```
template <class TC>
invoke_t<TC, expr_plus<int>> f(...)
{
    using namespace value_or_error:
    auto e1 = expr1(...);
    if ( failed(e1) )
        return failure_value(e1);
    auto& v1 = success_value(e1);
    auto e2 = expr2(...);
    if ( failed(e2) )
        return failure_value(e2);
    auto& v2 = success_value(e2);
    return make<TC>(expr_plus<int>(v1, v2));
}
```

A curiously repeated try pattern

While doing error propagation the following pattern appears quite often

```
    auto e1 = expr1(...);
    if ( value_or_error::failed(e1) )
        return value_or_error::failure_value(e1);
    auto& v1 = value_or_error::success_value(e1);
```

This is the reason d'être of the proposed `operator try` P0779R0. Note that either the *try-expression*` or the Coroutine TS *co_await-expression* could be customized for *ValueOrError* types. See the appendix for more information. With that we would be able to have either

```
expected<expr_plus<int>, error_code> f(...)
{
    auto v1 = co_await expr1(...);
    auto v2 = co_await expr1(...);
    return expr_plus<int>(v1, v2);
}
```

```
expected<expr_plus<int>, error_code> f(...)
{
    auto v1 = try expr1(...);
    auto v2 = try expr1(...);
    return expr_plus<int>(v1, v2);
}
```

and even more

```
expected<expr_plus<int>, error_code> f(...)
{
    return expr_plus<int>(try expr1(...), try expr1(...));
}
```

Others are suggesting to borrow `operator?` from Rust as an alternative to `operator try` for *ValueOrError* types.

```
expected<expr_plus<int>, error_code> f(...)
{
    return expr_plus<int>( expr1(...)?, expr1(...)?);
}
```

In any case, it seams clear that using `co_await` for optional and expected is disturbing and the code associated by the customization is more complex to optimize.

Error handling with *ValueOrError* types

While the *ValueOrError* customization for the *try-expression* or *co_await-expression* or or *try-expression* are enough to propagate the underlying error as such, the user needs at a given moment to recover or propagate a different error. Next we describe some these utilities that could help to do that.

A generic `value_or` function for *ValueOrError* types

We have `optional::value_or()` and `expected::value_or()` functions with a similar definition. This function can be defined in a generic way for *ValueOrError* types as follows

```
template <ValueOrError X, class T>
auto value_or(X&& x, T&& v)
{
    using namespace value_or_error;
    if ( succeeded(forward<X>(x)) )
        return success_value(move(x));
    return forward<T>(v);
}
```

A generic `value_or_throw` function for *ValueOrError* types

We have `optional::value()` and `expected::value()` functions with a similar definition, but returning a specific exception. It has been argued that the user need sometimes to throw a specific exception more appropriated to his context. We can define a function for *ValueOrError* types that allows to specify the exception to throw as follows

```
template <class Exception, ValueOrError X>
auto value_or_throw(X&& x)
{
    using namespace value_or_error;
    if ( succeeded(forward<X>(x)) )
        return success_value(move(x));
    throw Exception{failure_value(move(x))};
}
```

A generic `resolve` function for *ValueOrError* types

The previous function `value_or_throw` is a special case of error handling. We can have a more general one `resolve` that takes a function having as parameter the failure type.

```
template <ValueOrError X, class F>
auto resolve(X&& x, F&& f)
{
    using namespace value_or_error;
    if ( succeeded(forward<X>(x)) )
        return success_value(move(x));
    throw invoke(forward<F>(f), failure_value(move(x)));
}
```

With this definition `value_or` could be defined as

```
template <ValueOrError X, class T>
auto value_or(X&& x, T v)
{
    return resolve(forward<X>(x), [v](auto &&failure) {
        return v;
    });
}
```

and `value_or_throw` could be defined as

```
template <class E, ValueOrError X>
auto value_or_throw(X&& x)
{
    return resolve(forward<X>(x), [](auto &&failure) {
        throw E{failure};
    });
}
```

A generic `error_or` function for *ValueOrError* types

It has been argued that the error should be always available (that it should be a wide function) and that often there is a success value associated to the error. We have the proposed `status_value`, something like

```
struct status_value {
    E status;
    optional<T> opt_value;
};
```

The following code shows a use case

```
auto e = function();
switch (e.status)
    success: ....; break;
    too_green: ....; break;
    too_pink: ....; break;
```

With the current expected interface interface the user could be tempted to do

```
auto e = function();
if (e)
    /*success:*/ ....;
else
    switch (e.error())
    case too_green: ....; break;
    case too_pink: ....; break;
```

This could be done with the current interface as follows

```
auto e = function();
switch (error_or(e, success))
    success: ....; break;
    too_green: ....; break;
    too_pink: ....; break;
```

where

```
template <ValueOrError X, class E>
E error_or(X && x, E&& err) {
    using namespace value_or_error;
    if ( failed(forward<X>(x) )
        return failure_value(move(x));
    return forward<E>(err);
}
```

Need for *ValueOrError* `error`

Note that the previous `value_or` function works for `optional` and `expected` as both have a success type that match the value type. However, `error_or` doesn't works for `expected` as `expected<T,E>` is not implicitly convertible from `E` but from `unpexpected<E>` which wraps an `E`.

For *ValueOrError* types for which the success type wraps the value type and/or the failure type wraps the error type, we need to unwrap the `success` / `failure` types to get a `value` / `error` types respectively.

```
template <ValueOrError X, class T>
auto value_or(X&& x, T&& v)
{
    using namespace value_or_error;
    if ( succeeded(forward<X>(x) )
        return underlying(success_value(move(x)));
    return forward<T>(v);
}
```

If `wrapping::underlying` is the identity for non-wrapping types, we have that the previous definition works well for any *ValueOrError* types. Otherwise we need a `underlying_or_identity` function that is the identity except when the type is a *Wrapping* type.

For this *ValueOrError* types it will be better to define two functions that unwrap directly the success or the failure value

```
namespace value_or_error {
    // ...
    template <class X>
    auto deref(X&& x)
    {
        return underlying_or_identity(success_value(forward<X>(x)));
    }
    template <class X>
    auto error(X&& x)
    {
        return underlying_or_identity(failure_value(forward<X>(x)));
    }
}
```

and we can as well rename the `succeeded` / `failed` functions to be more inline with the `optional` / `expected` interface

```
namespace value_or_error {
    // ...
    template <class X>
    auto has_value(X && x)
    {
        return succeeded(forward<X>(x));
    }
    template <class X>
    auto has_error(X && x)
    {
        return failed(forward<X>(x));
    }

}
```

With these definitions we can have a more generic definition for `value_or` and `error_or`.

```
template <ValueOrError X, class T>
auto value_or(X&& x, T&& v)
{
    using namespace value_or_error;
    if ( has_value(forward<X>(x) )
        return deref(move(x)));
    return forward<T>(v);
}

template <ValueOrError X, class E>
E error_or(X && x, E&& err) {
    using namespace value_or_error;
    if ( has_error(forward<X>(x) )
        return error(move(x));
    return forward<E>(err);
}
```

A generic `check_error` function for *ValueOrError* types

Another use case which could look much uglier is if the user had to test for whether or not there was a specific error code.

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

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

```
e = function();
while ( check_error(e, timeout) )
{
    sleep(delay);
    delay *=2;
    e = function();
}
```

where

```
template <ValueOrError X, class E>
bool check_error(X && e, E&& err) {
    using namespace value_or_error;
    if ( has_value(forward<X>(x)) )
        return false;
    return error(forward<X>(x))) == forward<E>(err);
}
```

Functors and Monads

`functor::transform`

There is a natural way to apply a function to any *ValueOrError* given the function takes the *ValueOrError* value type as parameter when the *ValueOrError* is *TypeConstructible*. The result type will be a *ValueOrError* where the value type is return type of the function.

`monad::bind`

In the same way there is also a natural way to apply a monadic function to any *ValueOrError* given the function takes the *ValueOrError* value type as parameter and returns the same kind of *ValueOrError* with the same error type when the *ValueOrError* is *TypeConstructible*. The result type will be the result type of the function.

Proposal

This paper proposes

- to add *Wrapping* types that allows to unwrap a wrapping type to get its underlying value,
- to add *ValuedOrError* types with `succeeded(n)` / `has_value(n)` , `failed(n)` /has_error(n), `success_value(n)` , `failure_value(n)` , `deref(n)` and `error(n)` functions,
- to add *ValueOrNone* types as an extension of *Nullable* types that are not *NullablePointer* for which there is only a possible value type, adding the `deref(n)` function,
- to map *ValueOrNone* types to *ValuedOrError* types when we consider `none_type_t<T>` as the `failure_type` and the `error_type` ,

- customize the standard types `std::optional`, `std::experimental::expected` to these concepts,
- to add the following helper functions for *ValuedOrError* types
    - `value_or`,
    - `value_or_throw`,
    - `resolve`,
    - `error_or` and,
    - `check_error`.
- to add monadic functions when the type is *TypeConstructible*, and
- to map *ValuedOrError* types as *SumType* types by defining a `visit` function.

Design Rationale

Should `underlying` be the identity for any non-specialized *Wrapping* type?

In any case, we need to have access to the underlying value if wrapped and to the value itself otherwise. So either `underlying` does it or we can define another function that does it.

We have two options:

- `wrapping::traits<T>::underlying` is not defined by default and we define another function `underlying_or_identity`.
- `wrapping::traits<T>::underlying` is the identity by default and then we have already the function.

In order to identify clearly the *wrapping* types we have chosen to define an additional `underlying_or_identity` function, and let `underlying` for *Wrapping* types.

Do we need `success_value`?

Should we see *ValueOrError* as a sum type of `value_or_error::value_type` or `value_or_error::error_type` or a sum type of `value_or_error::success_type` or `value_or_error::failure_type`?

Note that we want to see `expected<T,E>` as the sum type of `T` and `unexpected<E>`.

`success_value` function has a sense only if we want the last.

While we don't propose yet a type for which `value_or_error::value_type` and `value_or_error::success_type` are different, we could have one *ValueOrError* type that wraps the the value type using `success<T>` and the `value_or_error::error_type` using `value_or_error::failure<E>`. This type wouldn't need to be implicitly convertible from the value type, but just for his `value_or_error::success_type`.

Customization

This proposal follows the drafted [CUSTOM] customization points approach, shared by most of my proposals. It can be adapted if required to the [N4381] customization points approach.

Wrapping types naming

We need a better name for the concept.

In Haskell these kind of types have a builtin construction, data types, like `Left l`, `Right r` or `Just t`. But Haskell doesn't provide a generic access to the UT of these types. It uses pattern matching.

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. There are however some classes in the standard that needs to be customized.

Proposed Wording

The proposed changes are expressed as edits to N4617 the Working Draft - C++ Extensions for Library Fundamentals V2, but pretend to go to the V3 TS.

This wording will be completed if there is an interest in the proposal.

Add the "Wrapping Objects" section

Wrapping Objects

A *Wrapping* type is a type that wraps an underlying type and provides access to the stored underlying value.

Header synopsis [Wrapping.synop]

```
namespace std::experimental {
inline namespace fundamentals_v3 {

namespace wrapping {

template <class T, class Enabler=void>
    struct traits;

template <class T>
    constexpr auto underlying(T && x);

template <class T>
    struct unwrapped_type;
template <class T>
    using unwrapped_type_t = typename unwrapped_type<T>::type;

template <class T>
struct unwrapped_type
{
    using type = remove_reference_t<decltype(wrapping::underlying(declval<T>()))>;
};

template <class E, when<is_enum_v<E>>>
    struct traits
    {
        template <class T>
            constexpr auto underlying(T && x)
            {
                return static_cast<std::unwrapped_type_t<T>>(forward<T>(x));
            }

    }

}

using wrapping::underlying;

template <class T>
struct is_wrapping;
template <class T>
struct is_wrapping<const T> : is_wrapping<T> {};
template <class T>
struct is_wrapping<volatile T> : is_wrapping<T> {};
template <class T>
struct is_wrapping<const volatile T> : is_wrapping<T> {};

template <class T>
constexpr bool is_wrapping_v = is_wrapping<T>::value;

template <class T>
    constexpr auto underlying_or_identity(T && x);

}}
```

Template class `wrapping::traits` [Wrapping.traits]

```
namespace wrapping {
    template <class T, class Enabler=void>
        struct traits;
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

Template function `wrapping::underlying` [Wrapping.underlying]

```
namespace wrapping {
    template <class T>
        constexpr auto underlying(T && x);
}
```

Effects: forward the call to the `traits<T>::underlying` .

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::succeeded(forwards<T>(v))` is well formed and return a type convertible to `bool` .

Template function `underlying_or_identity` [Wrapping.underlying<u>or</u>identity]

```
template <class T>
    constexpr auto underlying_or_identity(T && x);
}
```

Equivalent to: `wrapping::underlying(forward<T>(v))` if `is_wrapping_v<T>` , and `forward<T>(v)` otherwise.

Add a "ValueOrError Objects" section

ValueOrError Objects

Header synopsis [ValueOrError.synop]

```
namespace std::experimental {
inline namespace fundamentals_v3 {
namespace value_or_error {

template <class T, class Enabler=void>
    struct traits {};

template <class T> constexpr bool succeeded(T && v) noexcept;
template <class T> constexpr bool failed(T && v) noexcept;
template <class T> constexpr bool has_value(T && v) noexcept;
template <class T> constexpr bool has_error(T && v) noexcept;

template <class T> constexpr auto success_value(T&& x);
template <class T> constexpr auto failure_value(T&& x);

template <class T> struct success_type;
template <class T> using success_type_t = typename success_type<T>::type;

template <class T> struct failure_type;
template <class T> using failure_type_t = typename failure_type<T>::type;
```

```cpp
template <class T> constexpr auto deref(T&& x);
template <class T> constexpr auto error(T&& x);

template <class T> struct value_type;
template <class T> using value_type_t = typename value_type<T>::type;

template <class T> struct error_type;
template <class T> using error_type_t = typename error_type<T>::type;

}

template <class T> struct is_value_or_error;
template <class T> struct is_value_or_error <const T>
    : is_value_or_error <T> {};
template <class T> struct is_value_or_error <volatile T>
    : is_value_or_error <T> {};
template <class T> struct is_value_or_error <const volatile T>
    : is_value_or_error <T> {};

template <class T> constexpr bool is_value_or_error_v = is_value_or_error <T>::value ;

namespace value_or_error {

// when type constructible, is a functor
template <class T, class F> constexpr auto transform(T&& n, F&& f);

// when type constructible, is an applicative
template <class F, class T> constexpr auto ap(F&& f, T&& n);

// when type constructible, is a monad
template <class T, class F> constexpr auto bind(T&& n, F&& f);

// when type constructible, is a monad_error
template <class T, class F> constexpr auto catch_error(T&& n, F&& f);
template <class T, class ...Xs> constexpr auto make_error(Xs&&...xs);

// sum_type::visit
template <class N, class F> constexpr auto visit(N&& n, F&& f);

// helper functions
template <class N, class F>
    constexpr auto resolve(N&& n, F&& f);

template <class X, class T>
    constexpr auto value_or(X&& ptr, T&& val);

template <class E, class X>
    constexpr auto value_or_throw(X&& ptr);

template <class X, class E>
    constexpr auto error_or(X&& ptr, E&& err);

template <class X, class E>
    constexpr bool check_error(X&& n, E&& err);

}

}
```

```
}
```

Template class `value_or_error::traits` [valueorerror.traits]

```
namespace value_or_error {
    template <class T, class Enabler=void>
        struct traits {};
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

class `value_or_error::mcd_success_or_failure` [valueorerror.mcd]

Minimal complete definition based of `succeeded` , `success_value` or `failure_value` . Requires the types the be *wrapping*.

```
namespace value_or_error {
    struct mcd_success_or_failure
    {
        template <class U>
        static constexpr
        bool failed(U && ptr) noexcept;

        template <class U>
        static
        bool has_value(U && u);

        template <class U>
        static
        bool has_error(U && u);

        template <class U>
        static
        auto deref(U && u);

        template <class U>
        static
        auto error(U && u);
    };
}
```

Template function `value_or_error.mcd_success_or_failure::failed` [valueorerror.mcd.failed]

```
    template <class T>
        static bool failed(T && v) noexcept;
}
```

Equivalent to: `! value_or_error::succeeded(v)` .

Template function `has_value` [valueorerror.mcd.has_value]

```
    template <class T>
        static constexpr bool has_value(T && v) noexcept;
```

Equivalent to: `value_or_error::succeeded(v)`.

Template function `has_error` [valueorerror.mcd.has_error]

```
namespace value_or_error {
    template <class T>
        static constexpr bool has_error(T && v) noexcept;
}
```

Equivalent to: `! value_or_error::succeeded(v)`.

Template function `deref` [valueorerror.mcd.deref]

```
namespace value_or_error {
    template <class T>
        static constexpr auto deref(T&& x);
}
```

Equivalent to: `underlying_or_identity(success_value(v))`.

Template function `error` [valueorerror.mcd.error]

```
namespace value_or_error {
    template <class T>
        static constexpr auto error(T&& x);
}
```

Equivalent to: `underlying_or_identity(failure_value(v))`.

Template function `succeeded` [valueorerror.succeeded]

```
namespace value_or_error {
    template <class T>
        bool succeeded(T && v) noexcept;
}
```

Effects: forward the call to the `traits<T>::succeeded`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::succeeded(forwards<T>(v))` is well formed and return a type convertible to `bool`.

Template function `failed` [valueorerror.failed]

```
namespace value_or_error {
    template <class T>
        bool failed(T && v) noexcept;
}
```

Effects: forward the call to the `traits<T>::failed`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>:: failed(forwards<T>(v))` is well formed and return a type convertible to `bool`.

Template function `has_value` [valueorerror.has_value]

```
namespace value_or_error {
    template <class T>
        constexpr bool has_value(T && v) noexcept;
}
```

Effects: forward the call to the `traits<T>::has_value`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::has_value(forwards<T>(v))` is well formed and return a type convertible to `bool`.

Template function `has_error` [valueorerror.has_error]

```
namespace value_or_error {
    template <class T>
        constexpr bool has_error(T && v) noexcept;
}
```

Effects: forward the call to the `traits<T>::has_error`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::has_error(forwards<T>(v))` is well formed and return a type convertible to `bool`.

Template function `success_value` [valueorerror.success_value]

```
namespace value_or_error {
    template <class T>
        constexpr auto success_value(T&& x);
}
```

Effects: forward the call to the `traits<T>::success_value`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::success_value(forwards<T>(v))` is well formed.

Template function `failure_value` [valueorerror.failure_value]

```
namespace value_or_error {
    template <class T>
        constexpr auto failure_value(T&& x);
}
```

Effects: forward the call to the `traits<T>::failure_value`.

Remark: The previous function shall not participate in overload resolution unless:

- `traits<T>::failure_value(forwards<T>(v))` is well formed.

Add a "ValueOrNone Objects" section

ValueOrNone Objects

Header synopsis [ValueOrNone.synop]

```
namespace std::experimental {
inline namespace fundamentals_v3 {

    template <class T> struct is_value_or_none;

    template <class T>
        constexpr bool is_value_or_none_v = is_value_or_none <T>::value ;

    template <class T>
        struct is_value_or_none<const T> : is_value_or_none<T> {};
    template <class T>
        struct is_value_or_none<volatile T> : is_value_or_none<T> {};
    template <class T>
        struct is_value_or_none<const volatile T> : is_value_or_none<T> {};

namespace value_or_none {
    using nullable::has_value;
    using nullable::none;
    using nullable::none_type_t;

    // class traits
    template <class T>
        struct traits;

    template <class T> constexpr auto deref(T&& x);

    template <class T>
        struct value_type;
    template <class T>
        using value_type_t = typename value_type<T>::type;

    template <class T> constexpr auto deref_none(T&& );
}

namespace value_or_error
{
  template <class T>
  struct traits<T, meta::when<is_value_or_none<T>::value>>
        : mcd_success_or_failure
  {
      template <class U> static constexpr bool succeeded(U && u);

      template <class U> static constexpr auto success_value(U && u);

      template <class U> static constexpr auto failure_value(U && u);
  };
}

}
}
}
```

Template class `is_value_or_none` [valueornone.isvalueor_none]

```
    template <class T> struct is_value_or_none;
```

inherits from `true_type` if the class `T` is *Nullable* and not *NullablePointer* and has been configured with the `value_or_none::traits and from` false_type`otherwise.

Template class `value_or_none::traits` [valueornone.traits]

This template class must be specialized when the type is a `ValueOrNone` providing the `deref` customization point.

```
namespace value_or_none {
    template <class T>
        struct traits;
}
```

Template Function `value_or_none::deref` [valueornone.deref]

```
namespace value_or_none {
    template <class T> constexpr auto deref(T&& x);
}
```

Optional Objects

Add Specialization of *ValueOrNone* [optional.object.valueornone].

20.6.x *ValueOrNone* specialization

`optional<T>` is a model of *ValueOrNone*.

```
namespace value_or_none {
template <class T>
struct traits<optional<T>> {
        template <class U> static constexpr auto deref(U && ptr);
};
}
```

Expected Objects

Add Specialization of *wrapping* [unexpected.object.wrapping]

The type `unexpected` is a *wrapping* type, when the `underlying` is just the value

```
namespace wrapping
{
template <class E>
struct traits<unexpected<E>>
{
    template <class U>
    static constexpr
    auto underlying(U && u) => u.value();
};
}
```

Add Specialization of *ValueOrError* [expected.object.valueorerror]

```
#include <experimental/value_or_error>
...
namespace value_or_error
{
template <class T, class E>
struct traits<expected<T,E>> : mcd_success_or_failure
{
    template <class U>
        static constexpr
        bool succeeded(U && e) noexcept => e.has_value();

    template <class U>
        static constexpr
        auto success_value(U && e) => *e;

    template <class U>
        static constexpr
        auto failure_value(U && e) => unexpected(e.error());

    template <class U>
        static constexpr
        auto error(U && e) => e.error();
};
}
```

Implementability

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

See W_impl, VOE_impl and VON_impl.

Open points

The authors would like to have an answer to the following question if there is any interest at all in this proposal:

Do we want `optional<T>` to be a model of *ValueOrError*?

`optional<T>` has a not-a-value `nullopt_t` and can be seen as a sum type of `nullopt_t` and `T`. But should we consider `nullopt_t` as an error? This is maybe disputable.

This paper assumes that we can see `nullopt_t` as an error on the context where `optional<T>` is returned by a function. Doing it has the advantage of inheriting all the possible mappings of a *ValueOrError* indirectly.

If we consider `nullopt_t` doesn't represent an error we will need to do explicitly the mappings of `optional<T>` for *Functor*, *Applicative* and *Monad*. I suspect that in this case we wouldn't want to see `optional<T>` as a *MonadError*.

Do we need to reintroduce `expected::get_unexpected`?

LEWG requested the removal `expected::get_unexpected` as we can use instead `unexpected(e.error())`. However the main reason d'être of having this function was just to get a reference to the failure type associated to `expected`.

If we want to visit `expected<T, E>`, we need two types `T` and `unexpected<E>` and building `unexpected<E>` from `expected<T, E>` seam more expensive that just returning a reference to stored `unexpected<E>`.

Without this function, either the standard implementation of the customization for expected is a friend of expected and is able to get a reference to stored `unexpected<E>` or we ave that `value_or_error::failure_value` must return by value for `expected`, which is not optimal.

Do we want the explicit customization traits approach?

Alternatively we could use ADL and [] ...

Is *Wrapping* absolutely needed?

We can simplify the proposal just by requiring a more explicit mapping for `expected` and remove the minimal complete definition based on wrapping objects.

Do we want to start with a simpler proposal, extract the *Wrapping* part and see later how to mix *Wrapping* and *ValueOrError* later?

Enums are *Wrapping* types

We have already that C++ enums wraps in some way an underlying integral type.

We can specialize `wrapping::traits<Enum>::underlying`.

Wrapping naming

I would like to be able to use `underlying_type` instead of `unwrapped_type`. But this will need to change the core of the language, as `std::underlying_type<E>` works only for enums, which this proposal don't pretend to.

Alternatively we could have a to `std::wrapping::underlying_type`, but not a `std::underlying_type` that can be used with *Wrapping* types.

Do we need `success_value`?

Should we see *ValueOrError* as a sum type of `value_or_error::value_type` or `value_or_error::error_type` or a sum type of `value_or_error::success_type` or `value_or_error::failure_type`?

Note that we want to see `expected<T,E>` as the sum type of `T` and `unexpected<E>`.

`success_value` function has a sense only if we want the last.

While we don't propose yet a type for which `value_or_error::value_type` and `value_or_error::success_type` are different, we could have one *ValueOrError* type that wraps the the value type using `success<T>` and the `value_or_error::error_type` using `value_or_error::failure<E>`. This type wouldn't need to be implicitly convertible from the value type, but just for his `value_or_error::success_type`.

ValueOrError naming

`succeeded` versus `has_value`

`failed` versus `has_error`?

`success_value`?

Other alternatives

* `get_success`

`failure_value`?

* `get_failure`

`deref`?

`deref` is clearly not the good name.

Other alternatives

* `value`: `optional::value` and `expected::value` throw an exception if not valued.
* `get`: `future::get` follows the expected signature.
* `get_value`

`error`?

`expected::error` has this meaning.

Other alternatives

* `get_error`

Do we need *ValueOrNone*?

We can get rid of *ValueOrNone* and define a explicit specializations for `optional<T>`.

As we don't have yet other *ValueOrNone* in the standard, maybe this generalization can wait.

File(s) name

Should we include this in `<experimental/functional>` or in a specific file? We believe that a specific file is a better choice as this is needed in `<optional>` and `<experimental/expected>`. We propose to locate each concept in its one file `<experimental/wrapping>` / `<experimental/valued_or_error>` / `<experimental/valued_or_none>`.

About `value_or_error::value(n)`

We could define a wide `value_or_error::value(n)` function on *ValueOrError* that obtain the value or throws an exception. If we want to have a default implementation the function will need to throw a generic exception `bad_access`.

However to preserve the current behavior of `std::optional::value()` / `std::expected::value()` we will need to be able to consider this function as a customization point also.

The user can alternatively use `value_or_throw`, which allows to specify the exception.

Do we want a `value_or_error::value` function that throw `bad_access`?

Do we want a customizable `value_or_error::value`? Should the exceptions throw by this function inherit from a common exception class `bad_access`?

Future work

We have an implementation of the following, but we don't have wording yet.

ValueOrError as SumType

A *ValueOrError* can be considered as a sum type. It is always useful reflect the related types. `value_or_error::error_type_t` and `value_or_error::value_type_t` give respectively the associated non-a-value and the value types.

ValueOrError as a Functor

While we don't have yet an adopted proposal for *Functor*, we can define a default `value_or_error::transform` function for *ValueOrError* type when the type-constructor is *TypeConstructible*.

ValueOrError as an Applicative Functor

While we don't have yet an adopted proposal for *ApplicativeFunctor*, we can define a default `value_or_error::ap` function for *ValueOrError* when the type-constructor is *TypeConstructible*.

ValueOrError as a Monad

While we don't have yet an adopted proposal for *Monad*, we can define a default `value_or_error::bind` function for *ValueOrError* when the type-constructor is *TypeConstructible*.

ValueOrError as a MonadError

While we don't have yet an adopted proposal for *MonadError*, we can define a default `value_or_error::catch_error` and `value_or_error::make_error` functions for *ValueOrError* when the type-constructor is *TypeConstructible*.

Acknowledgements

Thanks to Niall for his idea of the `operator try` which motivated the definition of these concepts and for which a direct implementation is possible.

Thanks to Arthur O'Dwyer for the idea to restrict them to value types.

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

History

Revision 1

- Remove smart pointer as types modeling *ValueOrNone*, as pointers don't preserve.
- Rename *Wrapped* to *Wrapping* and `unwrap` to `underlying`.
- Improve the wording.
- Add mapping to *Functor*/*Monad*.

Revision 0

- Extract `deref()` / `visit()` and the derived algorithms as `value_or` and `error_or` from P0196R3 and define *ValueOrError*/*ValueOrNone*, as `std::any` cannot define `deref()` and `std::any` should be *Nullable*.

References

- N4381 Suggested Design for Customization Points

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4381.html

- 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

- P0050R0 C++ generic match function

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0050r0.pdf

- P0088R0 Variant: a type-safe union that is rarely invalid (v5)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0088r0.pdf

- P0091R0 Template parameter deduction for constructors (Rev. 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

- P0196R3 Generic `none()` factories for *Nullable* types (Rev. 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0196r3.html

- P0323R4 A proposal to add a utility class to represent expected monad

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r4.pdf

- P0338R2 C++ generic factories

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0338r2.pdf

- P0343R1 - Meta-programming High-Order functions

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0343r1.pdf

- P0779R0 Proposing operator try()

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0779r0.pdf

- CWG 1630 Multiple default constructor templates

  http://open-std.org/JTC1/SC22/WG21/docs/cwg_defects.html#1630

- SUM_TYPE Generic Sum Types

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

- W_impl Wrapped types

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

- VOE_impl ValueOrError types

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

- VON_impl ValueOrNone types

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

- CUSTOM An Alternative approach to customization points

  https://github.com/viboes/std-make/blob/master/doc/proposal/customization/customization_points.md