Document number:	p0786R0
Date:	2017-10-15
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 *ValeuOrNone*.

Types that are possibly valued and have a single error are named in this paper *ValedOrError*. 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` , pointers and smart pointers are *ValeuOrNone* types. The proposed `std::experimental::expected` P0323R4 is a *ValuedOrError* type.

Table of Contents

- Introduction
- Motivation and Scope
- Proposal
- Design Rationale
- Proposed Wording
- *Implementability
- Open points
- Acknowledgements
- History
- References

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 unwrap the wrapped value before calling to the user provided functions.

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() bool(ptr) *o *e *ptr nullopt e.error() nullptr nullopt unexpected(e.error()) nullptr	value_or_error::has_value(o) value_or_error::has_value(e) value_or_error::has_value(ptr) value_or_error::deref(o) value_or_error::deref(e) value_or_error::deref(ptr) value_or_error::error(o) value_or_error::error(e) value_or_error::error(ptr) value_or_error::failure_value(o) value_or_error::failure_value(e) value_or_error::failure_value(ptr)
Functor/Monad	
(o) ? f(*o) : nullopt (o) ? f(*o) : unexpected(e.error) (ptr) ? f(*ptr) : nullptr (o) ? g(*o) : nullopt (o) ? g(*o) : unexpected(e.error) (ptr) ? g(*ptr) : nullptr	value_or_error::transform(o, f) value_or_error::transform(e, f) value_or_error::transform(ptr, f) value_or_error::bind(e, g) value_or_error::bind(e, g)git mv value_or_error::bind(ptr, g)
Helper	
o.value_or(v) e.value_or(v) (ptr) ? *ptr : 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::value_or(ptr, v) value_or_error::error_or(e, err) value_or_error::check_error(e, err)

Motivation and Scope

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

values

```
optional<expr_plus<int>> f(...)
{
    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> f(...)
{
    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

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

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

Error handling with *ValueOrError* types

While the *ValueOrError* customization for the *try-expression* or *co_await-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 E, ValueOrError X>
auto value_or_throw(X&& x)
{
    using namespace value_or_error;
    if ( succeeded(forward<X>(x)) )
        return success_value(move(x));
    throw E{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 and that often there is a success value associated to the error. We have the `status_value` proposal and `expected<T,E>` could be seen more like something like the proposed

```
struct status_value {
    E error;
    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 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 type to get a value/error type.

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

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 wrapped::unwrap(success_value(forward<X>(x)));
    }
    template <class X>
    auto error(X&& x)
    {
        return wrapped::unwrap(failure_value(forward<X>(x)));
    }
}
```

and we can as well rename the succeed/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);
}
```

If `wrapped::unwrap` is the identity for non-wrapped types, we have that the previous definition works well for any *ValueOrError* types.

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 will 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 *Wrapped* types that allows to unwrap a wrapped type,
- 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 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`, smart pointers, `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

Customization

This proposal follows the drafted CUSTOM customization points approach. It can be adapted if required to the N4381 customization points approach.

Naming

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.

This paper depends in some way on the helper classes proposed in P0343R1, as e.g. the place holder `_t` and the associated specialization for the type constructors `optional<_t>`, `unique_ptr<_t>`, `shared_ptr<_t>`.

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.

Adapt the "ValueOrError Objects" section

ValueOrError Objects

Header synopsis [Wrapped.synop]

Header synopsis [ValueOrError.synop]

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

    // class traits
    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 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 TC>
        using failure_type_t = typename failure_type<TC>::type;

    template <class T>
        using success_type_t = decltype(success_value(declval<T>));
    template <class T>
        using success_type_t = decltype(failure_value(declval<T>));

    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 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 TC>
    using error_type_t = typename error_type<TC>::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);

}

}
}
```

class `traits` [valueorerror.traits]

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

    // class mcd_success_or_failure
    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
        auto deref(U && u);

        template <class U>
        static
        auto error(U && u)
        JASEL_DECLTYPE_RETURN_NOEXCEPT (
                wrapped::unwrap(value_or_error::failure_value(forward<U>(u)))
        )
    };

    // class traits specialization for pointers
    template <class T>
        struct traits<T*>
            : traits_pointer_like<T*>
        {};
}
```

Template function `succeeded` [valueorerror.succeeded]

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

Template function `failed` [valueorerror.failed]

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

Template function `has_value` [valueorerror.has_value]

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

Adapt the "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 namespace nullable;

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

    // class traits_pointer_like
    struct traits_pointer_like
    {
        template <class U> static constexpr auto deref(U && ptr);
    };

    // class traits specialization for pointers
    template <class T> struct traits<T*> : traits_pointer_like<T*> {};

    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);
  };
}

}
}
}
```

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>> : traits_pointer_like{};
}
```

Smart Pointers

20.6.x *ValueOrNone* specialization

`unique_ptr<T, D>` is a models of *ValueOrNone*.

```
namespace value_or_none {
    template <class T, class D>
    struct traits<unique_ptr<T, D> : traits_pointer_like {};
}
```

`shared_ptr<T>` is a models of *ValueOrNone*.

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

Expected Objects

Add Specialization of *Wrapped* [unexpected.object.wrapped].

```
namespace wrapped
{
template <class E>
struct traits<unexpected<E>>
{
    template <class U>
    static constexpr
    auto unwrap(U && u);

};
}
```

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

```
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;

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

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

Implementability

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

See 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 the explicit customization approach?

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 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 namming

`succeeded` versus `has_value`

`failed` versus `has_error`?

`success_value`?

`failure_value`?

`deref`?

`error`?

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/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.

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*.

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*.

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*.

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.

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

History

Revision 0

- Extract `deref()` / `visit()` and the derived alogorithms 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

- 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