

Document number:	P0196R4
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 >

Generic `none()` factories for *Nullable* types

Abstract

In the same way we have *NullablePointer* types with `nullptr` to mean a null value, this proposal defines *Nullable* requirements for types for which `none()` means the null value. This paper proposes some generic `none()` factories for *Nullable* types like `optional`, pointers and smart pointers.

Note that for *Nullable* types the null value doesn't mean an error, it is just a value different from all the other values, it is none of the other values.

Table of Contents

[Introduction](#) [Motivation and Scope](#) [Proposal](#) [Design Rationale](#) [Proposed Wording](#) [Implementability](#) [Open points](#) [Acknowledgements](#)
[History](#) [References](#)

Introduction

There are currently two adopted single-value (unit) types, `nullptr_t` for pointer-like classes and `nullopt_t` for `optional<T>`. [P0088R0](#) proposes an additional `monostate_t` as yet another unit type. Most languages get by with just one unit type. [P0032R0](#) proposed a new `none_t` and corresponding `none` literal for the class `any`. The feedback from the Kona meeting was that should not keep adding new “unit” types like this and that we need to have a generic `none` literal at least for non pointer-like classes.

Revision 0 for this paper presented a proposal for a generic `none_t` and `none` (no-value) factory, creates the appropriate not-a-value for a given *Nullable* type.

Revision 1 presented two kind of `none` factories `none()` and `none<T>()`.

Revision 2 makes it possible to consider pointer-like types a *Nullable*.

Revision 3 adds a new `nullable::deref` customization point and a lot of algorithms that can be built on top of *Nullable* thanks to this addition, as *Functor* `transform`, *ApplicativeFunctor* `ap`, *Monad* `bind`, *SumType* `visit`, and some minor algorithms `value_or` and `apply_or`.

Revision 4 rollbacks the *Nullable*s cannot provide always a `deref`. We have moved `deref` to another concept hierarchy [P0786R0](#). We will need to refactor this proposal in accordance to [P0786R0](#).

Having a common syntax and semantics for this factories would help to have more readable and teachable code, and potentially allows us to define generic algorithms that need to create such a no-value instance.

Note however that we would not be able to define interesting algorithms without having other functions around the *Nullable* concept as e.g. being able to create a *Nullable* wrapping instance containing the associated value (the make factory [P0338R2](#)) and observe the value or the not-a-value a *Nullable* type contains, or visitation type switch as proposed in [P0050R0](#), or the getter functions proposed in [P0042], or Functor/Monadic operations. This is left for future proposals.

BEFORE	AFTER
<p>Construction</p> <pre> int* p = nullptr; unique_ptr<int> sp = nullptr; shared_ptr<int> sp = nullptr; optional<int> o = nullopt; //unique_ptr<int> sp = unique_ptr{}; //shared_ptr<int> sp = shared_ptr{}; //optional<int> o = optional{}; any a = any{}; </pre>	<pre> int* p = none(); unique_ptr<int> sp = none(); shared_ptr<int> sp = none(); optional<int> o = none(); any a = none(); //int* p = none<add_pointer>(); shared_ptr<int> sp = none<shared_ptr>(); shared_ptr<int> sp = none<unique_ptr>(); optional<int> o = none<optional>(); any a = none<any>(); </pre>
<p>Conversion</p> <pre> void g(int*); void f(unique_ptr<int>); void f(optional<int>); void f(any); g(nullptr); f(nullptr); f(nullopt); //f(unique_ptr{}); //f(optional{}); f(any{}); </pre>	<pre> void g(int*); void f(unique_ptr<int>); void f(optional<int>); void f(any); //g(none<add_pointer>()); f(none<unique_ptr>()); f(none<optional>()); f(none<any>()); </pre>
<p>Return</p>	

```

template <template <class ...> class TC, class T>
TC<T> f(T) {
    return TC<T>{};
}

f<add_pointer_t>(a)
f<optional>(a)
f<unique_ptr>(a)
f<shared_ptr>(a)

```

```

template <template <class ...> class TC, class T>
invoke_t<quote<TC>,T> f(T) {
    return none<TC>();
}

//f<add_pointer>(a)
f<optional>(a)
f<unique_ptr>(a)
f<shared_ptr>(a)

template <class TC, class T>
invoke_t<TC,T> f(T) {
    return none<TC>();
}

f<add_pointer<_t>>(a)
f<optional<_t>>(a)
f<unique_ptr<_t>>(a)
f<shared_ptr<_t>>(a)
f<any>(a)

```

Motivation and Scope

Why do we need a generic `none()` literal factory

There is a proliferation of “unit” types that mean no-value type,

- `nullptr_t` for pointer-like objects and `std::function`,
- `std::nullopt_t` for `optional<T>`,
- `std::monostate` unit type for `std::variant<std::monostate_t, Ts...>` (in [P0088R0](#)),
- `none_t` for `any` (in [P0032R0](#) - rejected as a specific unit type for `any`)

Having a common and uniform way to name these no-value types associated to *Nullable* types would help to make the code more consistent, readable, and teachable.

A single overarching `none_t` type could allow us to define generic algorithms that operate across these generic *Nullable* types.

Generic code working with *Nullable* types, needs a generic way to name the null value associated to a specific *Nullable* type `N`. This is the reason d'être of `none<N>()`.

Possible ambiguity of a single no-value constant

Before going too far, let me show you the current situation with `nullptr` and to my knowledge why `nullptr` was not retained as no-value constant for `optional<T>` - opening the gates for additional unit types.

NullablePointer types

All the pointer-like types in the standard library are implicitly convertible from and equality comparable to `nullptr_t`.

```
int* ip = nullptr;
unique_ptr<int> up = nullptr;
shared_ptr<int> sp = nullptr;
if (up == nullptr) ...
if (ip == nullptr) ...
if (sp == nullptr) ...
```

Up to now everything is ok. We have the needed context to avoid ambiguities.

However, if we have an overloaded function as e.g. `print`

```
template <class T>
void print(unique_ptr<T> ptr);
template <class T>
void print(shared_ptr<T> ptr);
```

The following call would be ambiguous

```
print(nullptr);
```

Wait, who wants to print `nullptr` ? Surely nobody wants. Anyway we could add an overload for `nullptr_t`

```
void print(nullptr_t ptr);
```

and now the last overload will be preferred as there is no need to conversion.

If we want however to call to a specific overload we need to build the specific pointer-like type, e.g if wanted the `shared_ptr<T>` overload, we will write

```
print(shared_ptr<int>{});
```

Note that the last call contains more information than should be desired. The `int` type is in some way redundant. It would be great if we could give as less information as possible as in

```
print(nullptr<shared_ptr>);
```

Clearly the type for `nullptr<shared_ptr>` couldn't be `nullptr_t`, nor a specific `shared_ptr<T>`. So the type of `nullptr<shared_ptr>` should be something different, let me call it e.g. `nullptr_t<shared_ptr>`

You can read `nullptr<shared_ptr>` as the null pointer value associated to `shared_ptr`.

Note that even if template parameter deduction for constructors [P0091R0](#) is adopted we are not able to write the following, as the deduced type will not be the expected one.

```
print(shared_ptr(nullptr));
```

We are not proposing these for `nullptr` in this paper, it is just to present the context. To the authors knowledge it has been accepted that the user needs to be as explicit as needed.

```
print(shared_ptr<int>{});
```

Why `nullopt` was introduced?

Lets continue with `optional<T>`. Why the committee didn't wanted to reuse `nullptr` as the no-value for `optional<T>` ?

```
optional<int> oi = nullptr;  
oi = nullptr;
```

I believe that the two main concerns were that `optional<T>` is not a pointer-like type even if it defines all the associated operations and that having an `optional<int*>`, the following would be ambiguous

```
optional<int*> sp = nullptr;
```

We need a different type that can be used either for all the *Nullable* types or for those that are wrapping an instance of a type, not pointing to that instance. At the time, as the problem at hand was to have an `optional<T>`, it was considered that a specific solution will be satisfactory. So now we have

```
template <class T>  
void print(optional<T> o);  
  
optional<int> o = nullopt;  
o = nullopt;  
print(nullopt);
```

Moving to *Nullable* types

Some could think that it is better to be specific. But what would be wrong having a single way to name this no-value for a specific class using `none` ?

```
optional<int> o = none();  
any a = none();  
o = none();  
a = none();
```

So long as the context is clear there is no ambiguity.

We could as well add the overload to `print` the no-value `none`

```
void print(none_t);
```

and

```
print(none());  
print(optional<int>{});
```

So now we can see `any` as a *Nullable* if we provide the conversions from `none_t`

```
any a = none();
a = none();
print(any{});
```

Nesting *Nullable* types

We don't provide a solution to the following use case. How to initialize an `optional<any>` with an `std::any` `none()`

```
optional<any> oa2 = any{}; // assert(o)
optional<any> oa1 = none(); // assert(! o)
```

If we want that

```
optional<any> oa1 = none<any>(); // assert(o)
```

the resulting type for `none<any>()` shouldn't be `none_t` and we will need a `nullany_t`. This paper don't includes yet this `nullany_t`, but the author considers that this is the best direction. Have a common `none_t` that can be used when there is no ambiguity and `none<T>` to disambiguate.

Note that `any` is already `Nullable`, so how will this case be different from

```
optional<optional<int>> oo1 = optional<int>{};
optional<optional<int>> oo2 = nullopt;
```

or from nested smart pointers.

```
shared_ptr<unique_ptr<int>> sp1 = unique_ptr<int>{};
shared_ptr<unique_ptr<int>> sp2 = nullptr;
```

However we propose a solution when the result type of not-a-value of the two *Nullable*s is a different type.

```
optional<unique_ptr<int>> oup1 = none(); // assert(! o)
optional<unique_ptr<int>> oup1 = nullptr; // assert(o)

optional<unique_ptr<int>> oup1 = none<optional>; // assert(! o)
optional<unique_ptr<int>> oup1 = none<unique_ptr>; // assert(o)
```

The result type of `none<Tmpl>()` depends on the `Tmpl` parameter.

Other operations involving the unit type

There are other operations between the wrapping type and the unit type, such as the mixed equality comparison:

```
o == nullopt;
a == any{};
```

Type erased classes as `std::any` don't provide comparison.

However `Nullable` types wrapping a type as `optional<T>` can provide mixed comparison if the type `T` is ordered.

```
o > none()
o >= none()
!(o < none())
!(o <= none())
```

So the question is whether we can define these mixed comparisons once for all on a generic `none_t` type and a model of `Nullable`.

```
template < Nullable C >
bool operator==(none_t, C const& x) { return ! std::has_value(x); }
template < Nullable C >
bool operator==(C const& x, none_t) { return ! std::has_value(x); }
template < Nullable C >
bool operator!=(none_t, C const& x) { return std::has_value(x); }
template < Nullable C >
bool operator!=(C const& x, none_t) { return std::has_value(x); }
```

The ordered comparison operations should be defined only if the `Nullable` class is `Ordered`.

Differences between `nullopt_t` and `monostate_t`

`std::nullopt_t` is not `DefaultConstructible`, while `monostate_t` must be `DefaultConstructible`.

`std::nullopt_t` was required not to be `DefaultConstructible` so that the following syntax is well formed for an optional object

```
o
```

```
o = {}
```

So we need a `none_t` that is `DefaultConstructible` but that `{}` is not deduced to `nullopt_t{}`. This is possible if `nullopt_t` default constructor is explicit (See [LWG 2510](#), [CWG 1518](#) and [CWG 1630](#)).

The `std::experimental::none_t` is a user defined type that has a single value `std::experimental::none()`. The explicit default construction of `none_t{}` is equal to `none()`. We say `none_t` is a unit type.

Note that neither `nullopt_t`, `monostate_t` nor the proposed `none_t` behave like a tag type so that [LWG 2510](#) should not apply.

Waiting for [CWG 1518](#) the workaround could be to move the assignment of `optional<T>` from a `nullopt_t` to a template as it was done for `T`.

Differences between `nonesuch` and `none_t`

Even if both types contains the none word they are completely different. `std::experimental::nonesuch` is a bottom type with no instances and, `std::experimental::none_t` is a unit type with a single instance.

The intent of `nonesuch` is to represent a type that is not used at all, so that it can be used to mean not detected. `none_t` intent is to represent a type that is none of the other alternatives in the sum type.

`nullable::none_type_t` and `nullable::value_type_t`.

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

Proposal

This paper proposes to

- add `none_t / none()`,
- add `none<TC>()`, `none<Tmpl>()`,
- add requirements for *Nullable* and *StrictWeaklyOrderedNullable* types, and derive the mixed comparison operations on them,
- add some minor changes to `optional`, `any` to be constructed from `none_t` and to customize the *Nullable* requirements.

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++14. 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.

Add a "Nullable Objects" section

Nullable Objects

No-value state indicator

The `std::experimental::none_t` is a user defined type that has a factory `std::experimental::none()`. The explicit default construction of `none_t{}` is equal to `none()`. `std::experimental::none_t` shall be a literal type. We say `none_t` is a unit type.

[Note: `std::experimental::none_t` is a distinct unit type to indicate the state of not containing a value for *Nullable* objects. The single value of this type `none()` is a constant that can be converted to any *Nullable* type and that must equally compare to a default constructed *Nullable*. -- endnote]

Nullable requirements

A *Nullable* type is a type that supports a distinctive null value. A type `N` meets the requirements of *Nullable* if:

- `N` satisfies the requirements of *EqualityComparable*, *DefaultConstructible*, and *Destructible*,
- the expressions shown in the table below are valid and have the indicated semantics, and
- `N` satisfies all the other requirements of this sub-clause.

A value-initialized object of type `N` produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type `N` may have an indeterminate value. [Note: Operations involving indeterminate values may cause undefined behavior. — end note]

No operation which is part of the *Nullable* requirements shall exit via an exception.

In Table X below, `u` denotes an identifier, `t` denotes a non-const lvalue of type `N`, `a` and `b` denote values of type (possibly const) `N`, `x` denotes a (possibly const) expression of type `N`, and `nN` denotes `std::experimental::none<N>()` and `n` denotes `std::experimental::none()`.

Expression	Return Type	Operational Semantics
<code>nullable::none<N>()</code>	<code>none_type_t<N></code>	
<code>N{}</code>		post: <code>N{} == nN</code>
<code>N u(n)</code>		post: <code>u == nN</code>
<code>N u(nN)</code>		post: <code>u == nN</code>
<code>N u = n</code>		post: <code>u == nN</code>
<code>N u = nN</code>		post: <code>u == nN</code>
<code>N(n)</code>		post: <code>N(n) == nN</code>
<code>N(nN)</code>		post: <code>N(nN) == nN</code>
<code>std::has_value(x)</code>	contextually convertible to bool	true if <code>x != nN</code>
<code>a != b</code>	contextually convertible to bool	<code>!(a == b)</code>
<code>a == np, np == a</code>	contextually convertible to bool	<code>a == N{}</code>
<code>a != np, np != a</code>	contextually convertible to bool	<code>!(a == N{})</code>

***StrictWeaklyOrderedNullable* requirements**

A type `N` meets the requirements of *StrictWeaklyOrderedNullable* if:

- `N` satisfies the requirements of *StrictWeaklyOrdered* and *Nullable*.

Header synopsis [nullable.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
namespace nullable {

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

    // class none_t
    struct none_t;

    // none_t relational operators
    constexpr bool operator==(none_t, none_t) noexcept;
    constexpr bool operator!=(none_t, none_t) noexcept;
    constexpr bool operator<(none_t, none_t) noexcept;
    constexpr bool operator<=(none_t, none_t) noexcept;
    constexpr bool operator>(none_t, none_t) noexcept;
    constexpr bool operator>=(none_t, none_t) noexcept;

```

```

// none_t factory
constexpr none_t none() noexcept;

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

// class traits_pointer_like
struct traits_pointer_like;

// class traits specialization for pointers
template <class T>
    struct traits<T*>;

template <class T>
    constexpr auto none() -> `see below` noexcept;

template <template <class ...> class TC>
    constexpr auto none() -> `see below` noexcept;

template <class T>
    using none_type_t = decltype(nullable::none<T>());
}

using nullable::none_t;
using nullable::none_type_t;
using nullable::none;
using nullable::has_value;

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

template <class T>
    constexpr bool is_nullable_v = is_nullable<T>::value ;

template <class T>
    struct is_strict_weakly_ordered_nullable;

namespace nullable {
    // Comparison with none_t
    template < class C >
        bool operator==(none_t, C const& x) noexcept;
    template < class C >
        bool operator==(C const& x, none_t) noexcept;
    template < class C >
        bool operator!=(none_t, C const& x) noexcept;
    template < class C >
        bool operator!=(C const& x, none_t) noexcept;

    template < class C >
        bool operator<(none_t, C const& x) noexcept;

```

```

template < class C >
    bool operator<(C const& x, none_t) noexcept;
template < class C >
    bool operator<=(none_t, C const& x) noexcept;
template < class C >
    bool operator<=(C const& x, none_t) noexcept;
template < class C >
    bool operator>(none_t, C const& x) noexcept;
template < class C >
    bool operator>(C const& x, none_t) noexcept;
template < class C >
    bool operator>=(none_t, C const& x) noexcept;
template < class C >
    bool operator>=(C const& x, none_t) noexcept;

}}}}

```

No-value state indicator [nullable.none_t]

The struct `none_t` is an empty structure type used as a unique type to indicate the state of not containing a value for *Nullable* objects. It shall be a literal type.

```

namespace nullable {
    struct none_t{
        explicit none_t() = default;
        template <class T>
            operator T*() const noexcept { return nullptr; }
    };
}

```

`none_t` relational operators [nullable.none_t.rel]

```

namespace nullable {
    constexpr bool operator==(none_t, none_t) noexcept { return true; }
    constexpr bool operator!=(none_t, none_t) noexcept { return false; }
    constexpr bool operator<(none_t, none_t) noexcept { return false; }
    constexpr bool operator<=(none_t, none_t) noexcept { return true; }
    constexpr bool operator>(none_t, none_t) noexcept { return false; }
    constexpr bool operator>=(none_t, none_t) noexcept { return true; }
}

```

[Note: `none_t` objects have only a single state; they thus always compare equal. — end note]

`none_t` factory [nullable.none_t.fact]

```

namespace nullable {
    constexpr none_t none() noexcept { return none_t{}; }
}

```

class `traits` [nullable.traits]

```

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

    // class traits_pointer_like
    struct traits_pointer_like
    {
        static constexpr
            nullptr_t none() noexcept { return nullptr; }
        template <class Ptr>
            static constexpr
                bool has_value(Ptr ptr) { return bool(ptr) }
    };

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

```

Template function `has_value` [possiblynotvalued.has_value]

```

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

```

Template function `none` [nullable.none]

```

namespace nullable {
    template <class T>
        constexpr auto none() ->
            decltype(nullable::traits<T>::none()) noexcept;

    template <template <class ...> class TC>
        constexpr auto none() ->
            decltype(none<type_constructor_t<meta::quote<TC>>>>()) noexcept;
}

```

Template class `is_nullable` [nullable.is_nullable]

```

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

template <class T>
    constexpr bool is_nullable_v = is_nullable<T>::value ;

```

Template class `is_strict_weakly_ordered_nullable`

[[nullable.isstrictweaklyorderednullable](#)]

```

template <class T>
    struct is_strict_weakly_ordered_nullable :
        conjunction<is_strict_weakly_ordered<T>, is_nullable<T>> {};

```

Nullable comparison with `none_t` [[nullable.none_t_ops](#)]

```

namespace nullable {
    template < class C >
        bool operator==(none_t, C const& x) noexcept
            { return ! ::std::has_value(x); }
    template < class C >
        bool operator==(C const& x, none_t) noexcept
            { return ! ::std::has_value(x); }
    template < class C >
        bool operator!=(none_t, C const& x) noexcept
            { return ::std::has_value(x); }
    template < class C >
        bool operator!=(C const& x, none_t) noexcept
            { return ::std::has_value(x); }
}

```

Remark: The previous functions shall not participate in overload resolution unless `C` satisfies `* Nullable*`.

StrictWeaklyOrderedNullable comparison with `none_t` [[nullable.none_t_ord_ops](#)]

```

template < class C >
    bool operator<(none_t, C const& x) noexcept
        { return ::std::has_value(x); }
template < class C >
    bool operator<(C const& x, none_t) noexcept
        { return false; }
template < class C >
    bool operator<=(none_t, C const& x) noexcept
        { return true; }
template < class C >
    bool operator<=(C const& x, none_t) noexcept
        { return ! ::std::has_value(x); }
template < class C >
    bool operator>(none_t, C const& x) noexcept
        { return false; }
template < class C >
    bool operator>(C const& x, none_t) noexcept
        { return ::std::has_value(x); }
template < class C >
    bool operator>=(none_t, C const& x) noexcept
        { return ! ::std::has_value(x); }
template < class C >
    bool operator>=(C const& x, none_t) noexcept { return true; }
}

```

Remark: The previous functions shall not participate in overload resolution unless `C` satisfies *StrictWeaklyOrderedNullable*.

Optional Objects

Add conversions from `none_t` in [optional.object].

```

template <class T> class optional {
// ...
// 20.6.3.1, constructors
constexpr optional(none_t) noexcept;

// 20.6.3.3, assignment
optional &operator=(none_t) noexcept;
};

```

Update [optional.object.ctor] adding before p 1.

```
constexpr optional(none_t) noexcept;
```

Update [optional.object.assign] adding before p 1.

```
optional<T>& operator=(none_t) noexcept;
```

Add Specialization of *Nullable* [optional.object.nullable].

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

```
namespace nullable {
    template <class T>
    struct traits<optional<T>> : traits_pointer_like{
        static constexpr
        nullopt_t none() noexcept { return nullopt; }
    };
}
```

Class Any

Add conversions from `none_t` in `[any.object]`.

```
class any {
    // ...
    // 20.7.3.1, construction and destruction
    constexpr any(none_t) noexcept;

    // 20.7.3.2, assignments
    any &operator=(none_t) noexcept;
};
```

Update `[any.cons]` adding before p 1.

```
constexpr any(none_t) noexcept;
```

Effects: As if `reset()`

Postcondition: `this->has_value() == false`.

Update `[any.assign]` adding after p 12.

```
any<T>& operator=(none_t) noexcept;
```

Effects: As if `reset()`

Returns: `*this`

Postcondition: `has_value() == false`.

Add Specialization of *Nullable* `[any.object.nullable]`.

20.6.x Nullable

`any` is a model of *Nullable*.

```

namespace nullable {
    template <>
    struct traits<any> {
        template <class A>
        static constexpr
        bool has_value(A && v) noexcept { return v.has_value(); }
        static constexpr
        none_t none() noexcept { return none_t{}; }
    };
}

```

Variant Objects

Add Specialization of *Nullable* [any.object.nullable].

x.y.z Nullable

`variant<none_t, Ts...>` is a models of *Nullable*.

```

namespace nullable {
    template <class ...Ts>
    struct traits<variant<none_t, Ts...>> {
        static constexpr
        none_t none() noexcept { return none_t{}; }
        template <class U>
        static constexpr
        bool has_value(U && v) noexcept { return v.index()>0; }
    };
}

```

Smart Pointers

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

```

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

```

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

```

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

```

Implementability

This proposal can be implemented as pure library extension, without any language support, in C++14. However the adoption of [CWG 1518](#), [CWG 1630](#) makes it simpler.

Open points

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

- Should we include `none_t` in `<experimental/functional>` or in a specific file?
 - We believe that a specific file is a better choice as this is needed in `<experimental/optional>`, `<experimental/any>` and `<experimental/variant>`. We propose `<experimental/nullable>`.
- Should the mixed comparison with `none_t` be defined implicitly?
 - An alternative is to don't define them. In this case it could be better to remove the *Nullable* and *StrictWeaklyOrderedNullable* requirements as the "reason d'être" of those requirements is to define these operations.
- Should *Nullable* require in addition the expression `n = {}` to mean reset?
- Should `std::any` be considered as *Nullable*? Note that `std::any` is not *EqualityComparable*. Should we relax the *Nullable* requirements?
- Should we add `nullany_t` type as the `none_type_t<any>` to avoid ambiguities?.
- Should `variant<none_t, Ts ...>` be considered as *Nullable*?

Acknowledgements

Thanks to Tony Van Eerd and Titus Winters for helping me to improve globally the paper. Thanks to Agustín Bergé K-ballo for his useful comments. Thanks to Ville Voutilainen for the pointers about explicit default construction.

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

History

Revision 4

- Rollback the modifications of revision 3, as they belong to *ValueOrError* [P0786R0](#) kind of types.
 - We should move `deref()` to *ValueOrNone*, as `std::any` cannot define it, as it has not a unique value. As consequence the functions added in the previous version `value_or` and `apply_or` should be moved as well.
 - The same applies to the added `visit` function. We cannot visit any *Nullable* as we don't know what are its possibly values. This function belongs to *ValueOrNone* and to *ValueOrError*. *ValueOrNone* is a *Nullable* that has only one possible value type. *ValueOrNone* is specialization of *ValueOrError* where the Error is a unit type.
 - Next follows the concept hierarchy:
 - *SuccessOrValue* (succeeded, successvalue, failurevalue) (examples result => TRY
 - *ValueOrError* (has_value, deref, error) (examples: expected, result => SumType, Functor, Applicative, Monad, MonadError
 - *ValueOrNone*
 - *Nullable* (hasvalue, none) (examples std::variant<nonet, Ts...>, `std::any`)
 - *ValueOrNone* (deref) (examples T*, `std::optional`, `std::unique_ptr`, ...) => *ValueOrError*

- Move *Functor*, *Applicative*, *Monad*, *MonadError* specialization to *ValueOrError* [P0786R0](#).

Revision 3

Added the following specific *Nullable* functions and types to see *Nullables* as *Functors*, *Applicatives*, *Monads* and *SumType*:

- Added customization point `nullable::deref`.
- Added `nullable::none_type_t` and `nullable::value_type_t`.
- Added `nullable::transform` (*Functor*), `nullable::ap` (*Applicative*) and `nullable::bind` (*Monad*) [P0650R0](#) when the *Nullable* is also *TypeConstructible* [P0338R2](#).
- Added `nullable::visit` (*sumtype*) `[SUMTYPE]`.
- Added `nullable::value_or` and `nullable::apply_or`.

Revision 2

Fixes some typos and take in account the feedback from Oulu meeting. Next follows the direction of the committee:

- Add more examples in the documentation, including nesting of *Nullables*.
- More explicit tests in the implementation.
- Pointers should be *Nullable*.
- `has_value` should be non-member.
- Added a before/after comparison table.

Unfortunately initializing the nested *Nullables* with a nested `none` is not possible if the associated none-type are the same. This is in line with `optional<optional<T>>`.

Other changes:

- Consider having `none_type<T>` traits derived from the `none<T>()` function.
- Consider adding `is_nullable` type trait and `nullable::tag`.
- `std::any` cannot be considered as *Nullable* as far as we request *EqualityComparable* as we do for *NullablePointer*.
- Add examples using Template argument deduction for constructors.

Revision 1

The 1st revision of [\[P0196R0\]](#) fixes some typos and takes in account the feedback from Jacksonville meeting. Next follows the direction of the committee: the explicit approach `none<optional>` should be explored.

The approach taken by this revision is to provide both factories but instead of a literal we use a functions `none()` and `none<optional>()`.

Revision 0

This takes in account the feedback from Kona meeting [P0032R0](#). The direction of the committee was:

- Do we want `none_t` to be a separate paper?

```
SF F N A SA
11 1 3 0 0
```

- Do we want the `operator bool` changes? No, instead a `.something()` member function (e.g. `has_value`) is

preferred for the 3 classes. This doesn't mean yet that we replace the existing explicit `operator bool` in `optional`.

- Do we want emptiness checking to be consistent between `any` / `optional` ? Unanimous yes

Provide <code>operator bool</code> for both	Y: 6 N: 5
Provide <code>.something()</code>	Y: 17 N: 0
Provide <code>=={}</code>	Y: 0 N: 5
Provide <code>==std::none</code>	Y: 5 N: 2
<code>something(any/optional)</code>	Y: 3 N: 8

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>

- [P0032R0](#) Homogeneous interface for variant, any and optional

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.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>

- [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>

- [P0650R0](#) C++ Monadic interface

<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0650r0.pdf>

- [P0786R0](#) *ValuedOrError* and *ValueOrNone* types

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0786r0.html>

- [LWG 2510](#) Tag types should not be DefaultConstructible

<http://cplusplus.github.io/LWG/lwg-active.html#2510>

- [CWG 1518](#) Explicit default constructors and copy-list-initialization

http://open-std.org/JTC1/SC22/WG21/docs/cwg_active.html#1518

- [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/viboestd-make/tree/master/include/experimental/fundamental/v3/sum_type