A generic `none_t` literal type 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 a generic `none` literal for `Nullable` types like `optional` and `any` and proposes to replace the variant `monostate_t` defined in `P0088R0` by `none_t`.

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.

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

This paper presents a proposal for a generic `none_t` and `none` (no-value) factory, creates the appropriate not-a-value for a given `Nullable` type.

Having a common syntax and semantics for this literal 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. been able to create a `Nullable` wrapping instance containing the associated value (the make factory `MAKEF`) and observe whether this `Nullable` type contains a value or not (e.g. a visitation type switch as proposed in [P0050], or the getter functions proposed in [P0042], or Functor/Monadic operations). This is left for a future proposal.

Motivation and Scope

Why do we need a generic `none` literal

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

- `nullptr_t` for pointer-like objects and `std::function`,

```cpp
none_t
```
std::experimental::nullopt_t for optional<T>,
std::experimental::monostate unit type for variant<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 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. This is the reason d'être of none_t and none.

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.

```cpp
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

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

The following call would be ambiguous
Wait, who wants to print `nullptr`? Surely nobody. Anyway we could add an overload for `nullptr_t`:

```cpp
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:

```cpp
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:

```cpp
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](https://isocpp.org/std/deduction-guide) is adopted we are not able to write as the deduced type will not be the expected one:

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

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.

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

**Why `nullopt` was introduced?**

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

```cpp
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

```cpp
template <class T>
void print(optional<T> o);
```

```cpp
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`?

```cpp
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`

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

and

```cpp
print(none);
print(optional<int>{});
```
So now we can see `any` as a `Nullable` if we provide the conversions from `none_t`

```cpp
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 `any` `none`

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

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

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

or from nested smart pointers.

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

**Other operations involving the unit type**

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

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

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

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

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

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

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

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

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

**Differences between `nullopt_t` and `monostate_t`**

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

`std::experimental::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 that a `none_t` that is `DefaultConstructible` but that `{}` is not deduced to `nullopt_t{}`. This is possible if `nullopt_t` default constructor is explicit and `CWG 1518` is adopted.

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`.

\[ o > \text{none} \\
\[ o >= \text{none} \\
\[ !\left( o < \text{none} \right) \\
\[ !\left( o <= \text{none} \right) \]
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 product type or that can be stored in `any`.

Proposal

This paper proposes to

- add `none_t` / `none`
- add requirements for `Nullable` and `StrictWeaklyOrderedNullable` types, and derive the mixed comparison operations on them,
- add some minor changes to `optional`, `any` and `variant` to take `none_t` as their no-value type.

Proposed Wording

The proposed changes are expressed as edits to N4564 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 single value `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 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 below, \( u \) denotes an identifier, \( t \) denotes a non-const lvalue of type \( N \), \( x \) denotes a (possibly const) expression of type \( N \), and \( n \) denotes a value of type (possibly const) \( \text{std::experimental::none}_t \).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return Type</th>
<th>Operational Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N\ u(n) )</td>
<td></td>
<td>post: ( u == N{} )</td>
</tr>
<tr>
<td>( N\ u = n )</td>
<td></td>
<td>post: ( u == N{} )</td>
</tr>
<tr>
<td>( t = n )</td>
<td>( N&amp; )</td>
<td>post: ( t == N{} )</td>
</tr>
<tr>
<td>( x\text{.has_value()} )</td>
<td>contextually convertible to bool</td>
<td>( x != N{} )</td>
</tr>
</tbody>
</table>

Mixed equality comparison between a Nullable and a \( \text{none}_t \) are defined as

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

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

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

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

**StrictWeaklyOrderedNullable requirements**

A type \( N \) meets the requirements of StrictWeaklyOrderedNullable if:

- \( N \) satisfies the requirements of StrictWeaklyOrdered and Nullable.

Mixed ordered comparison between a StrictWeaklyOrderedNullable and a \( \text{none}_t \) are defined as
template < StrictWeaklyOrderedNullable C >
bool operator<(none_t, C const& x) { return x.has_value(); }
template < StrictWeaklyOrderedNullable C >
bool operator<(C const& x, none_t { return false; }

template < StrictWeaklyOrderedNullable C >
bool operator<=(none_t, C const& x) { return true; }
template < StrictWeaklyOrderedNullable C >
bool operator<=(C const& x, none_t { return !x.has_value(); }

template < StrictWeaklyOrderedNullable C >
bool operator>(none_t, C const& x) { return false; }
template < StrictWeaklyOrderedNullable C >
bool operator>(C const& x, none_t { return x.has_value(); }

template < StrictWeaklyOrderedNullable C >
bool operator>=(none_t, C const& x) { return !x.has_value(); }
template < StrictWeaklyOrderedNullable C >
bool operator>=(C const& x, none_t { return true; }

Header synopsis [nullable.synop]
namespace std {
    namespace experimental {
        inline namespace fundamentals_v3 {

            struct none_t{
                explicit none_t() {}
            };

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

            constexpr none_t none = none_t();

            // Comparison with none_t
            template <Nullable C>
            bool operator==(none_t, C const& x) noexcept { return !x.has_value(); }
            template <Nullable C>
            bool operator==(C const& x, none_t) noexcept { return x.has_value(); }
            template <Nullable C>
            bool operator!=(none_t, C const& x) noexcept { return x.has_value(); }
            template <Nullable C>
            bool operator!=(C const& x, none_t) noexcept { return x.has_value(); }
            template <StrictWeaklyOrderedNullable C>
            bool operator<(none_t, C const& x) { return false; }
            template <StrictWeaklyOrderedNullable C>
            bool operator<(C const& x, none_t) { return true; }
            template <StrictWeaklyOrderedNullable C>
            bool operator<=(none_t, C const& x) { return true; }
            template <StrictWeaklyOrderedNullable C>
            bool operator<=(C const& x, none_t) { return true; }
            template <StrictWeaklyOrderedNullable C>
            bool operator>(none_t, C const& x) { return false; }
            template <StrictWeaklyOrderedNullable C>
            bool operator>(C const& x, none_t) { return false; }
            template <StrictWeaklyOrderedNullable C>
            bool operator>=(none_t, C const& x) { return !x.has_value(); }
            template <StrictWeaklyOrderedNullable C>
            bool operator>=(C const& x, none_t) { return x.has_value(); }
        }
    }
}
Optional Objects

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

Add `optional<T>` is a model of `StrictWeaklyOrderedNullable` if `T` is a model of `StrictWeaklyOrdered`.

Remove the definition of `nullopt_t / nullopt`.

Replace any use of `nullopt_t / nullopt` by `none_t / none`.

Remove the mixed operations as redundant [optional.nullops].

Class Any

Add `any` is a model of `Nullable`.

Add a constructor from `none_t` equivalent to the default constructor.

Add a assignment from `none_t` equivalent assigning a default constructed object.

Variant Objects

Waiting for a specific wording for `variant` in a TS.

Remove the definition of `monostate_t`.

Replace any additional use of `monostate_t` by `none_t`.

Implementability

This proposal can be implemented as pure library extension, without any language support, in C++14. However the adoption of CWG 1518 will make 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` 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
I propose \texttt{<experimental/none>}. Should the mixed comparison with \texttt{none_t} be defined implicitly?

- An alternative is to don't define them. In this case it could be better to remove the \texttt{Nullable} and \texttt{StrictWeaklyOrderedNullable} requirements as the "reason d'être" of those requirements is to define these operations.

- Should \texttt{Nullable} require in addition the expression \texttt{n = \{} \texttt{to mean reset?}

- Should \texttt{variant<none_t, Ts ...>} be considered as \texttt{Nullable}?

  - This will need the addition of \texttt{v.has_value()}.  

- Bike-shading - \texttt{Nullable} versus \texttt{NullableValue}

### Acknowledgements

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

### References

- \texttt{N4564} N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS
  

- \texttt{P0032R0} Homogeneous interface for variant, any and optional
  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.pdf

- \texttt{P0091R0} Template parameter deduction for constructors (Rev. 3)
  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

- \texttt{P0088R0} Variant: a type-safe union that is rarely invalid (v5)
  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0088r0.pdf

- \texttt{MAKEF} C++ generic factories
  

- \texttt{LWG 2510} Tag types should not be DefaultConstructible
Appendix

Lifting  none

Not proposed by this paper, there is a possibility to lift none. This lifted value would express explicitly that the wrapped value would be used to emplace the optional wrapped valued.

```cpp
optional<any> o = lift(none);
```

The result of lift would be a type that will wrap none_t. optional<T> will need to accept a conversion from this lifted<U> by emplacing the type T from the type U.

Dependent  no_value<T>

A different on going proposal MAKEF C++ generic factories would propose a no_value function so that we could do

```cpp
optional<any> o = no_value<any>();
```

```cpp
shared_ptr<unique_ptr<int>> o = no_value<unique_ptr>();
```