| | |
|---|---|
| Document number: | D0196R0 |
| Date: | 2016-01-21 |
| Project: | ISO/IEC JTC1 SC22 WG21 Programming Language C++ |
| Audience: | Library Evolution Working Group |
| Reply-to: | Vicente J. Botet Escriba <vicente.botet@wanadoo.fr> |

# 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 .something()          Y: 17 N: 0
  Provide =={}                  Y:  0 N: 5
  Provide ==std::none           Y:  5 N: 2
  something(any/optional)       Y:  3 N: 8
```

# Table of Contents



# 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`,
- `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 unittypes.

## *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. 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 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 didn't the committee want 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 it 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 `any` none

```
optional<any> = none;
```

```
optional<any> = any{};
```

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

```
optional<optional<int>> = optional<int>{};
```

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

```
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`.

## 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::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.

```
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 ! 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.

# Proposal

This paper proposes to

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

## But how exactly is `none` defined?

This proposal doesn't impose a specific implementation. The `std::experimental::none` is a user defined literal that is the single value of the type, `std::experimental::none_t` . We say that `none_t` is a unit type.

A possible implementation could be based on the current definition of `std::experimental::nullopt_t` and `std::experimental::nullopt` .

# Proposed Wording

The proposed changes are expressed as edits to N4564 the Working Draft - C++ Extensions for Library Fundamentals V2.

# Nullable Objects

## No-value state indicator

The `std::experimental::none` is a user defined literal that is the single value of the type, `std::experimental::none_t`. We say that `none_t` is a unit type.

`std::experimental::none_t` shall be a literal type. Constant `none` shall be initialized with an argument of literal 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 distictive 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 subclause.

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) `std::experimental::none_t`.

| Expression | Return Type | Operational Semantics |
|---|---|---|
| N u(n) | | post: u == N{} |
| N u = n | | post: u == N{} |
| t = n | N& | post: t == N{} |
| x.has_value() | contextualy convertible to bool | x != N{} |

Mixed equality comparaison between a *Nullable* and a `none_t` are defined as

```
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 comparaison between a *StrictWeaklyOrderedNullable* and a `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_v2 {

    none // unspecified constant;
    using none_t = decltype(none);

    // 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 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; }
  }
  }
}
```

## Optional Objects

**Add** `optional<T>` is a model of *NullableValue*.

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

**Remove** the definiton of `nullopt_t` / `nullopt`.

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

**Remove** the mixed operations as redondant [optional.nullops].

## Class Any

**Add** `any` is a model of *NullableValue*.

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

**Add** a assignemnt from `none_t` equivalent assigning a default constructed object.

## Variant Object

Waiting for a specific wording in the TS.

**Remove** the definiton 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.

# Open points

The authors would like to have an answer to the following points if there is at all an interest 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 `<experimental/variant>`. I propose `<experimental/none>`.
- 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.

# Acknowledgements

Thanks to Tony Van Eerd for championing this proposal during the C++ standard committee meetings and helping me to improve globally the paper. Thanks to Agustín Bergé K-ballo for his useful comments.

# References


- N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.pdf

- P0032R0 Homogeneous interface for variant, any and optional

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

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

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

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

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

- MAKEF C++ generic factories

  https://github.com/viboes/std-make/blob/master/doc/proposal/factories/DXXXX_factories.md