`nullopt_t` and `nullptr_t` should both have `operator<=>` and `operator==`

Document #:	D2405R0
Date:	2021-07-09
Project:	Programming Language C++
Audience:	LEWG and EWG

Reply-to: Justin Bassett (jbassett271 at gmail dot com)

Abstract

`nullopt_t` can be three-way compared to `optional`. However, because there is no `operator<=>` or `operator==` between `nullopt_t`s, `optional` is not *`comparison_relation`*`_with nullopt_t` where *`comparison_relation`*`_with` is any of `equality_comparable_with`, `totally_ordered_with`, or `three_way_comparable_with`. Adding a trivial `operator<=>` for `nullopt_t` allows *`comparison_relation`*`_with` to support `optional + nullopt_t`. The same holds true with `nullptr_t` and `unique_ptr<T>` and `shared_ptr<T>`.

Contents

1 Motivation

1.1 Specific Usage Changes

These are some specific examples of code which this paper will simplify:

Before	After

// optional<T> range
`auto remove_nulls(auto& range) {`
`  return std::ranges::remove_if(`
`  range,`
`  [](const auto& v) {`
`  return v == std::nullopt;`
`  });`
`}`
	`auto remove_nulls(auto& range) {`
	`  return std::ranges::remove(`
	`  range, std::nullopt);`
	`}`

// shared_ptr<T> range
`auto after_null_sorted(auto& range) {`
`  return std::ranges::upper_bound(`
`  range,`
`  std::ranges::range_value_t<`
`  decltype(range)>(nullptr));`
`}`
	`auto after_null_sorted(auto& range) {`
	`  return std::ranges::upper_bound(`
	`  range, nullptr);`
	`}`

1.2 `nullopt_t`

It is trivial to define homogenous comparison operations for `nullopt_t`, as any singleton set is trivially strongly ordered by taking the single element to be equal to itself. Because `nullopt_t` is a singleton type and therefore meets the mathematical models, adding these comparison operations will not hide any logic errors, but it never makes sense to write `nullopt == nullopt` in ordinary code so it seems strange to add them. However, types should not be considered in isolation. `nullopt_t` should be considered in the context of `optional<T>`.

We have the ability to compare `optional<T>` and `nullopt_t` through `operator==(optional<T>, nullopt_t)` and `operator<=>(optional<T>, nullopt_t)`. However, without the comparison operators for `nullopt_t` itself, although we have `equality_comparable<optional<T>>`, `three_way_comparable<optional<T>>`, and `totally_ordered<optional<T>>`, we do not have the cross-type variants *`comparison_relation`*`_with<optional<T>, nullopt_t>`. This is because these variants include the requirements *`comparison_relation`*`<A>` and *`comparison_relation`*`<B>`, but `nullopt_t` does not satisfy any of these *`comparison_relation`*s because it has no comparison operators at all. Despite being irrelevant for `nullopt_t` on its own, `operator==` and `operator<=>` should be added to `nullopt_t` to fix this inconsistency with `optional<T>`.

1.3 `nullptr_t`

The argument for `nullptr_t` is the same as that as for `nullopt_t` except for `unique_ptr` and `shared_ptr`. It might at first appear that adding comparison operators for `nullptr_t` only makes sense in the context of `T*` where ordering comparisons are unspecified behavior, but `unique_ptr` and `shared_ptr` have custom ordering comparisons with `nullptr_t` which use `less<T*>` to produce a valid ordering. As such, these orderings should be defined for `nullptr_t` so that *`comparison_relation`*`_with<smart_ptr<T>, nullptr_t>` can be syntactically met.

2 Design Intent

In short, for both singleton types `nullptr_t` and `nullopt_t`, the same comparison operations should be valid:

— `nullopt <=> nullopt` should be `strong_ordering::equal`.

— `nullopt == nullopt` should be `true`.

— `nullopt != nullopt` should be `false`.

— `nullopt < nullopt` should be `false`.

— `nullopt > nullopt` should be `false`.

— `nullopt <= nullopt` should be `true`.

— `nullopt >= nullopt` should be `true`.

And similarly for `nullptr`, except note that `nullptr` already has equality operations defined.

For the case of `nullopt`, this can be easily accomplished by providing a defaulted `operator<=>`. For the case of `nullptr`, this requires defining `nullptr <=> nullptr` in [expr.spaceship] as well as the relational operators in [expr.rel] for consistency with other fundamental types.

2.1 Unresolved Issues

Even with this change, these *`comparison_relation_with`* concepts do not work with move-only types. For example `equality_comparable_with<optional<T>, nullopt_t>` for move-only `T` is still false. This issue will be resolved by [P2404R0].

3 Proposed wording

In [optional.nullopt]:

```
struct nullopt_t{see below};

struct nullopt_t{
  see below

  strong_ordering operator<=>(const nullopt_t&) const noexcept = default;
};

inline constexpr nullopt_t nullopt(unspecified);
```

In [expr.spaceship]:

> If both operands are of type `std::nullptr_t`, the result is of type `std::strong_ordering`. The result is `std::strong_ordering::equal`.
>
> Otherwise, the program is ill-formed.

In [expr.rel]:

> The converted operands shall have arithmetic, enumeration, ~~or~~ pointer type, or type `std::nullptr_t`. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.
>
> If both operands (after conversions) are of type `std::nullptr_t`, the relational operators yield results according to the following:
>
> — `nullptr < nullptr` yields `false`.
>
> — `nullptr > nullptr` yields `false`.
>
> — `nullptr <= nullptr` yields `true`.
>
> — `nullptr >= nullptr` yields `true`.

The proposed changes are relative to N4860 [Smith2020].

Document history

— **R0**, 2021-07-09 : Initial version.

References

[P2404R0] Justin Bassett. Relaxing equality_comparable_with's and three_way_comparable_with's common reference requirements to support move-only types. https://wg21.link/p2404r0, 2021.

[Smith2020] Richard Smith. Working Draft, Standard for Programming Language C++. https://isocpp.org/files/papers/N4860.pdf, 2020 (accessed 2021-07-07).