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

Document #: P2405R0
Date: 2021-07-15
Project: Programming Language C++
Audience: LEWG, EWG, and SG9

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_ts, 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 and 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

`auto remove_nulls(`
`  vector<optional<int>>& range)`
`{`
`  return ranges::remove(`
`    range, optional<int>());`
`}`
	`auto remove_nulls(`
	`  vector<optional<int>>& range)`
	`{`
	`  return ranges::remove(`
	`    range, nullopt);`
	`}`

`template <`
`  ranges::forward_range R>`
`  requires requires(`
`    ranges::range_value_t<R> val) {`
 	
`    // Require range of optional<T>`
`    requires same_as<`
`      decltype(val),`
`      decltype(optional(val))`
`    >;`
`  }`
`auto remove_nulls(R& range) {`
`  return ranges::remove(range,`
`    ranges::range_value_t<R>());`
`}`
	`template <`
	`  ranges::forward_range R>`
	`  requires requires(`
	`    ranges::range_value_t<R> val) {`
	
	`    // Require range of optional<T>`
	`    requires same_as<`
	`      decltype(val),`
	`      decltype(optional(val))`
	`    >;`
	`  }`
	`auto remove_nulls(R& range) {`
	`  return ranges::remove(`
	`    range, nullopt);`
	`}`

`auto after_null_sorted(`
`  vector<shared_ptr<int>>& range)`
`{`
`  return ranges::upper_bound(`
`    range, shared_ptr<int>());`
`}`
	`auto after_null_sorted(`
	`  vector<shared_ptr<int>>& range)`
	`{`
	`  return ranges::upper_bound(`
	`    range, nullptr);`
	`}`

`template <`
`  ranges::random_access_range R>`
`  // Assuming R is a range of some`
`  // smart_ptr<T>`
`auto after_null_sorted(R& range)`
`{`
`  return ranges::upper_bound(range,`
`    ranges::range_value_t<R>());`
`}`
	`template <`
	`  ranges::random_access_range R>`
	`  // Assuming R is a range of some`
	`  // smart_ptr<T>`
	`auto after_null_sorted(R& range)`
	`{`
	`  return ranges::upper_bound(`
	`    range, nullptr);`
	`}`

Note that some may reach for `ranges::algorithm_if` or `algorithm` instead:

```
// Instead of:
ranges::remove(range, ranges::range_value_t<R>());
// One of these may be used:
ranges::remove_if(range, [](const auto& o) { return o == nullopt; });
remove(range.begin(), range.end(), nullopt);
```

In fact, note that every unconstrained algorithm supports this use-case; only the constrained algorithms reject `nullopt` and `nullptr`.

As another example not concerning ranges, consider:

```
template <typename T>
class custom_set {
public:
  bool insert(T val);

  // Support heterogeneous lookup:
  template <std::totally_ordered_with<T> U>
  bool contains(const U& val);
};
```

Before	After

`bool has_null(`
`  const custom_set<`
`    shared_ptr<T>>& set)`
`{`
`  return set.contains(shared_ptr<T>());`
`}`
	`bool has_null(`
	`  const custom_set<`
	`    shared_ptr<T>>& set)`
	`{`
	`  return set.contains(nullptr);`
	`}`

1.2 Why is this useful, given that `optional<T>()` and `smart_ptr<T>()` work?

It is true that this issue can be worked around by replacing `nullopt` with `optional<T>()` and `nullptr` with `smart_ptr<T>()`, perhaps where those concrete types are computed through some type alias. Furthermore, optimizers consistently eliminate these temporaries, generating the same code either way. However, it is still beneficial to enable the usage `nullopt` and `nullptr`. `nullopt` and `nullptr` can be more readable than the constructor calls, as they clearly communicate their null value in their name. Furthermore, the same argument can be applied to the heterogeneous comparison operators we already have: why do we need heterogeneous comparison operators if we can simply use `optional<T>()` and `smart_ptr<T>()` in place of `nullopt` and `nullptr`? The issue with that argument is that these comparison operators are quite reasonable, as `opt == optional<T>(nullopt)` and `ptr == smart_ptr<T>(nullptr)` compile fine, so it is natural and consistent to be able to use `opt == nullopt` and `ptr == nullptr` as well. Given that we have these heterogeneous comparison operators, disallowing their use with constrained algorithms or constrained functions is inconsistent.

1.3 `nullopt_t`

It is trivial to define homogenous comparison operations for `nullopt_t`, as any singleton set is 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. Despite this, `nullopt == nullopt` never makes sense in ordinary code, as it could be replaced with `true`, making the idea of adding comparison operators seem illogical. 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>`.

1.4 `nullptr_t`

The argument for `nullptr_t` is the same as that as for `nullopt_t` except for `unique_ptr` and `shared_ptr`. The idea of comparison operators for `nullptr_t` initially brings to mind the context of `T*`, where relational comparisons only form a partial ordering that cannot compare null with other pointers without undefined behavior, but this forgets that `nullptr` is also the singleton null value for `unique_ptr` and `shared_ptr`. The smart pointers have custom relational comparisons with `nullptr_t` that use `less<T*>` to produce a valid total ordering. As such, these comparisons should be defined for `nullptr_t` so that *`comparison_relation`*`_with<smart_ptr<T>, nullptr_t>` can be syntactically met.

2 Background

2.1 `nullptr`'s historic relational operators

`nullptr` used to have relational comparisons and not just equality operators. However, [N3478] removed these `nullptr` comparisons as part of resolving `p > nullptr` given `T* p`, where `p > nullptr` was always undefined behavior, so removing this comparison operator turns a runtime bug into a compilation error. Without the context of the *`comparison_relation`*`_with` concepts, it seems obvious to remove the meaningless-in-isolation `nullptr`-only comparisons when removing `p > nullptr` regardless of the fact that the `nullptr` with `nullptr` comparisons do not have this same issue. Now that we have the *`comparison_relation`*`_with` concepts, we have a reason to add back in `nullptr`-only comparisons; a reason which does not conflict with the original reason that these comparisons were removed from the language.

Note that this paper does not propose adding comparison operators for any null pointer constructs other than `nullptr` itself. This means that `nullptr < (T*)nullptr` will not be made valid by this proposal, nor will `nullptr < (T*)0`. This apparent inconsistency is for a particular reason: directly writing `nullptr < nullptr` is not expected to appear in useful code. Instead, `nullptr` comparisons are expected to appear either through generic code or through concept syntactic requirements, where `nullptr` being of the special type `std::nullptr_t` is significant.

2.2 Why do the *`comparison_relation`*`_with<T, U>` concepts require *`comparison_relation`*`<T>` and *`comparison_relation`*`<U>`?

Cross-type equality must be carefully defined in mathematics. Equalities are equivalence relations, not just the `operator==(A, B)`. As equivalence relations are defined for a single set, cross-type equality is defined over a common supertype of A and B. That is, we take $C = A \cup B$ and define our equivalence relation over C, meaning that $\forall c_1, c_2 \in C, c_1 == c_2$ must be well-defined. Thus, as we could have $c_1, c_2 \in A$, $c_1, c_2 \in B$, or $c_1 \in A$ but $c_2 \in B$, so our equivalence relation must be defined for $A \times A$, $A \times B$, and $B \times B$. Translating to C++, `operator==(A, A)`, `operator==(A, B)`, `operator==(B, B)`, and `operator==(C, C)` must all be defined and be part of the same equivalence relation for us to have high confidence that the `operator==(A, B)` represents an actual equality. This is why we require `equality_comparable<A>` and `equality_comparable<B>`: to verify that the `operator==(A, B)` models equality.

The mathematics is the same for each of the other comparison relations.

3 Design Intent

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`.

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; `nullptr_t` already has equality operators defined.

3.1 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].

4 Proposed wording

In [optional.nullopt]:

```
struct nullopt_t{see below};
```

```
struct nullopt_t{
  see below

  friend constexpr strong_ordering operator<=>(nullopt_t, nullopt_t) 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 arithmetic or enumeration type, each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.
>
> If both operands (after conversions) are of type `std::nullptr_t`, the result is `true` if the operator is `<=` or `>=` and `false` otherwise.

The proposed changes are relative to the current working draft [N4878].

Document history

— **R0**, 2021-07-15 : Initial version.

References

[N3478] Jens Maurer. Core Issue 1512: Pointer comparison vs qualification conversions. `https://wg21.link/n3478`, 2012 (accessed 2021-07-09).

[N4878] Thomas Köppe. Working Draft, Standard for Programming Language C++. `https://wg21.link/n4878`, 2020 (accessed 2021-07-10).

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