Relaxing *comparison_relation*_with's common reference requirements to support move-only types

Document #:	D2404R0
Date:	2021-07-10
Project:	Programming Language C++
Audience:	LEWG and SG9

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

Abstract

Each *comparison_relation*_with—where *comparison_relation*_with is any of the concepts equality_comparable_with, totally_ordered_with, or three_way_comparable_with—does not support move-only types, because the common reference requirement requires that const T& and const U& are convertible to the possibly-not-a-reference common_reference_t. This common reference requirement should be relaxed to the mathematical ideal of a common *supertype* requirement, as the original reason to require formable references no longer exists and relaxing this requirement allows us to support move-only types.

Contents

Contents 1

1 Motivation . . . 1
2 Background . . . 4
3 Design . . . 5
4 Proposed wording . . . 7

References 11

1 Motivation

1.1 Overview

The common reference requirements of the *comparison_relation*_with concepts are stricter than the mathematical requirement. Ideally, this requirement could be relaxed to be as close to the mathematical requirement as possible to allow the maximum number of eligible types to satisfy these concepts.

For example, equality_comparable_with<unique_ptr<T>, nullptr_t> is false despite the fact that the heterogeneous operator== captures an actual equality. This happens because the common

reference requirement requires that the types are `convertible_to` the common reference, but `common_reference_t<const unique_ptr<T>&, const nullptr_t&>` is `unique_ptr<T>`, meaning that it requires `convertible_to<const unique_ptr<T>&, unique_ptr<T>>`, which is the same as requiring that `unique_ptr<T>` is copyable. The other direction is also possible, where `common_reference_t<const T&, const U&>` is `T` and a constructor `T(const U&)` does not exist but `T(U&&)` does exist. Because it has the same common reference requirement, this also applies to `three_way_comparable_with`.

1.2 Specific Code Changes

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

```
class bigint {
public:
  bigint(int);

  bigint(const bigint&) = delete;
  bigint(bigint&&) noexcept = default;
  bigint& operator=(const bigint&) = delete;
  bigint& operator=(bigint&&) noexcept = default;

  std::strong_ordering operator<=>(const bigint&) const;
  bool operator==(const bigint&) const;

  std::strong_ordering operator<=>(int) const;
  bool operator==(int) const;
};

class copy_bigint {
public:
  copy_bigint(bigint);

  std::strong_ordering operator<=>(const copy_bigint&) const;
  bool operator==(const copy_bigint&) const;

  std::strong_ordering operator<=>(const bigint&) const;
  bool operator==(const bigint&) const;
};
```

Before	After

`auto remove_zeros(`
`  vector<bigint>& range)`
`{`
`  return ranges::remove_if(`
`  range, [](const auto& i) {`
`  return i == 0;`
`  });`
`  // Alternatively:`
`  return remove(`
`  range.begin(), range.end(), 0);`
`}`
	`auto remove_zeros(`
	`  vector<bigint>& range)`
	`{`
	`  return ranges::remove(range, 0);`
	`}`

`auto find_sorted(`
`  vector<bigint>& range, int x)`
`{`
`  return ranges::lower_bound(`
`  range, x,`
`  less()); // NOT ranges::less`
`  // Alternatively:`
`  return lower_bound(`
`  range.begin(), range.end(), 0);`
`}`
	`auto find_sorted(`
	`  vector<bigint>& range, int x)`
	`{`
	`  return ranges::lower_bound(range, x);`
	`}`

`bool is_same(`
`  const vector<bigint>& lhs,`
`  const vector<copy_bigint>& rhs)`
`{`
`  return ranges::equal(`
`  lhs, rhs,`
`  // NOT ranges::equal_to`
`  equal_to());`
`  // Alternatively:`
`  return equal(`
`  lhs.begin(), lhs.end(),`
`  rhs.begin(), rhs.end());`
`}`
	`bool is_same(`
	`  const vector<bigint>& lhs,`
	`  const vector<copy_bigint>& rhs)`
	`{`
	`  return ranges::equal(lhs, rhs);`
	`}`

`bool multiset_includes(`
`  const vector<bigint>& lhs,`
`  const vector<copy_bigint>& rhs)`
`{`
`  return ranges::includes(`
`  lhs, rhs,`
`  less()); // NOT ranges::less`
`  // Alternatively:`
`  return includes(`
`  lhs.begin(), lhs.end(),`
`  rhs.begin(), rhs.end());`
`}`
	`bool multiset_includes(`
	`  const vector<bigint>& lhs,`
	`  const vector<copy_bigint>& rhs)`
	`{`
	`  return ranges::includes(lhs, rhs);`
	`}`

Notably, all of the above on the "After" column would compile today if `bigint` was copyable instead of move-only, although no copies will be made. Also, note that although all of the above examples use ranges, this issue would appear at any location where the *`comparison_relation`*`_with` concepts are used.

2 Background

2.1 Overview

`equality_comparable_with<T, U>` does far more than test for a compatible `operator==(T, U)`, instead attempting to capture true cross-type equality. To do so, it considers the equality in the context of a common supertype, codified as the requirement `common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&>`, which includes requiring both requirements `convertible_to<const T&, common_reference_t<const T&, const U&>>` and symmetrically `convertible_to<const U&, common_reference_t<const T&, const U&>>`. Because it is possible for `common_reference_t<const T&, const U&>` to be a non-reference type, these `convertible_to` requirements can end up requiring that we copy the `const T&` or `const U&`, especially if the `common_reference_t` is `T` or `U` itself as it is for the case of `unique_ptr<T>` and `nullptr`.

Importantly, the conversion to the common reference never needs to happen at runtime, as we can always use the provided heterogeneous `operator==(T, U)` instead. Historically, this was not the case, as the C++0X concepts had a mechanism that would resolve the `EqualityComparable<T, U>` cross type equality `t == u` as first converting to the common type if there was no heterogeneous `operator==(T, U)` [Stroustrup2012, 51]. However, as concepts are now only a way to check syntactic validity, this feature was removed.

`three_way_comparable_with` has the same common reference requirement and can similarly be relaxed. `totally_ordered_with` has this common reference requirement, but only transitively through `equality_comparable_with`.

2.2 Why the common reference requirement?

Cross-type equality is not initially well defined in mathematics, so some work must be done to capture it. The Palo Alto report [Stroustrup2012, 16] describes this conundrum. In particular, establishing an equivalence relation between two arbitrary sets A and B only makes sense if you instead establish the equivalence relation over $A \cup B$. In C++, this means that we need to think of the equality as operating over some common "supertype" of `T` and `U`. This requirement is codified in `equality_comparable_with` by the common reference requirement `common_reference_with`:

```
template<class T, class U>
  concept equality_comparable_with =
    equality_comparable<T> && equality_comparable<U> &&
    common_reference_with<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&> &&
    equality_comparable<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    weakly-equality-comparable-with<T, U>;
```

[N4878, 546]

Where `common_reference_with<T, U>` is defined as follows:

```
template<class T, class U>
  concept common_reference_with =
    same_as<common_reference_t<T, U>, common_reference_t<U, T>> &&
    convertible_to<T, common_reference_t<T, U>> &&
    convertible_to<U, common_reference_t<T, U>>;
```

[N4878, 540]

This requirement is not the same as the purely mathematical supertype requirement as C++ has to deal with objects and references, incidentally adding the requirement that this common reference must be formable from the two types.

This same argument applies to `three_way_comparable_with`: the relations only make sense when we lift the types to the common supertype, but this common supertype conversion never needs to happen at runtime. `three_way_comparable_with` similarly encodes this with the same invocation of `common_reference_with`.

3 Design

3.1 Overview

The problem with `equality_comparable_with` and `three_way_comparable_with` lies in the encoding of the supertype requirement as a common *reference* requirement; we want to encode the supertype requirement without requiring formable references or any particular cvref qualities. Considering `equality_comparable_with<T, U>` with the type `common_reference_t<const T&, const U&>` notated as `C`, this issue can be considered in two parts:

1. `T` is a move-only type, and `C` is the same as `T`.
2. `C` is not `T` and can only be constructed by an rvalue `T`.

For both of these issues, it is essential to note that although a conversion to `C` must exist to satisfy our mathematical axioms, we never need to perform this conversion, as we will always use the heterogeneous `operator==(T, U)`. This means that it is okay to make it require extreme acrobatics or even make it impossible to write a `bool equal_by_common(T, U)` function.

The first case can be solved by noting that, although the cvref-quality differs, `T` and `C` are of the same base type, so we can solve it by relaxing the `convertible_to<const T&, C>` requirement to also accept cases where `const T&` and `C` are the same after `remove_cvref_t`, which can be accomplished by using `convertible_to<const T&, const C&>` (and similarly for `U`). This works because if `const T&` is already `const C&`, we can simply bind the reference, but we can still construct a `C` from the `const T&` by binding the `const C&` to the temporary `C` object. Despite how dangerous that sounds, the risk is resolved by the fact that we do not have to do this at runtime.

The second case can be solved by relaxing the `convertible_to<const T&, C>` to not copy the `T`, but instead look for any valid conversion, which can be accomplished by using `convertible_to<T&&, C>` (and similarly for `U`).

Taking both solutions together yields `convertible_to<T&&, const C&>`, and this combined solution does not invalidate any of the prior arguments. Notably, these same arguments work for the comparisons encoded in `three_way_comparable_with`.

3.2 Syntactic requirements changes

Changing the meaning of `common_reference_with` is not the best idea, as the proposed changes are inconsistent with the concept's name. As such, it makes sense to add a new exposition only concept *`common-comparison-supertype-with`*`<T, U>` which applies these modifications to `common_reference_with`. However, since `T` and `U` are possibly cvref qualified, this new concept will also need to account for that by stripping the cvref qualifiers. `const` and references are mathematically meaningless, so stripping the cvref qualifiers does nott cause issue with the meaning of this exposition only concept.

In summary, *common-comparison-supertype-with*<T, U> is a variant of common_reference_with<remove_cvref_t<T>, remove_cvref_t<U>> which modifies the convertible_to<...> requirements to support move-only types.

This modified exposition only concept will replace the common_reference_with requirements in three_way_comparable_with and equality_comparable_with, transitively applying to totally_ordered_with as well.

3.3 Semantic requirements changes

Changing the syntactic requirements also requires that we change the semantic requirements of all of these concepts. Rather than purely copying the semantic requirements of common_reference_with where we construct the common reference via C(t) and C(u), *common-comparison-supertype-with* must instead capture the idea that we are moving to a const& by using static_cast<const C&>(move(t)).

For equality_comparable_with, the common supertype requirement may now move its arguments, but equality_comparable_with<T, U> specifies its semantic requirements using t and u of const remove_reference_t<T> and const remove_reference_t<U> respectively. Instead of having t and u be const, this paper proposes making them the non-const remove_cvref_t<T> and remove_cvref_t<U>, allowing us to move from t and u. This is not to prohibit the equality comparison of const lvalues, but the behavior of equality comparison of const lvalues must be the same as if they were non-const and moved from. Furthermore, despite moving from these lvalues, the objects should retain the exact same state as before they were moved from, because a move never actually happens at runtime. That is to say, the bool result of the heterogeneous operator== must be the same as if we move to the const C& common supertype and perform the comparison there, ignoring any side effects caused by the move. The same holds true for three_way_comparable_with and totally_ordered_with.

Actually encoding this new model is a bit tricky, because the comparison operators do not introduce a sequence point between their arguments. As such, the two comparisons must be evaluated in separate lines of code to prevent the move from affecting the heterogeneous comparison.

3.4 Potential issues with this approach

There are some issues with this approach:

— Changing any standard library concept is a breaking change for many reasons.

— Subsumption between equality_comparable_with and common_reference_with will be lost, and similarly with three_way_comparable_with.

— Changing convertible_to<const T&, C> to convertible_to<T&&, C> breaks for types C where C(const T&) exists, but C(T&&) is deleted or otherwise disabled.

The former two concerns should not block this proposal. The proposed changes should only help good code do what it is trying to do, with most breaks happening in almost pathological code. Breaks caused by the loss in subsumption should manifest themselves noisily, so the change in subsumption should be okay.

The last concern needs more consideration. Although some argue that this is misguided, a somewhat common pattern is to delete an rvalue overload of an overload set in attempt to prevent the function from being called with temporaries. In this instance, the overload set in consideration is the constructor, for which the deleted rvalue overload may be less common than arbitrary functions. Moreover, this pertains only to types which are the common_reference_t of other types, which are even less likely to manifest this constructor deletion. Despite the argued improbability, there is

an example of a type with a deleted rvalue overload in the standard library: `reference_wrapper`. However, `reference_wrapper` is not a natural common reference to be selected via specialization of `basic_common_reference`, and selecting it via the natural mechanisms usually leads to a situation where this concern does not manifest because other conversions are possible. That said, a potential fix for this issue would be to change the `convertible_to<T&&, C>` to `convertible_to<T&&, C> || convertible_to<const T&, C>`, allowing a fallback on the cross-type copy.

3.5 A smaller alternative which solves part of the problem

If we only wish to solve the first of the two issues referenced in the overview (3.1), the change to support this case would be significantly smaller. In particular, this issue is solved solely by modifying the syntactic requirement that `const T&` and `const U&` are convertible to the common reference `C` to instead additionally allow them to be the same as `C` after `remove_cvref_t`, requiring only the exposition only concept *`common-comparison-supertype-with`* with `convertible_to<const T&, const C&>` and similarly for `U`. The semantic requirements of this exposition only concept, `equality_comparable_with`, and `three_way_comparable_with` must still be modified, but only to the extent of replacing the constructor calls `C(t)` and `C(u)` with a `static_cast` which avoids calling the constructor if `T` or `U` are already the same type—barring cvref—as `C`: `static_cast<const C&>(t)` and `static_cast<const C&>(u)`.

3.6 Could we remove the common reference requirement?

It has been brought up a few times that perhaps we could remove the common reference requirement altogether, possibly by also requiring additional semantic requirements. This is an infeasible direction because a large number of types—including in the standard library—use `operator==` for something other than equality, so either these types would syntactically meet `equality_comparable_with` and just not actually work correctly, or we would have to have an explicit opt-in, barring a significant number of types from being `equality_comparable_with` when they trivially are. Furthermore, it is exceedingly easy to write an `operator==(T, U)` which feels like equality and even could be equality but actually is not when considered in the context of all of `operator==(T, T)`, `operator==(T, U)`, `operator==(U, U)`, and `operator==(C, C)` (where `C` is the common reference). To be a proper equality, all of these `operator==`s must be part of the same equality, otherwise we lose key properties of an equivalence class.

As an example, iterators and sentinels have a cross-type `operator==(iterator, sentinel)` which feels like equality and indeed could form an equivalence class, except that `operator==(iterator, iterator)` is *not* part of the same equivalence relation as `operator==(iterator, sentinel)`. Indeed, if these were to be part of the same equivalence relation, then `operator==(iterator, iterator)` must instead be testing to see if both iterators have reached the end of the range. Therefore, `equality_comparable_with<iterator, sentinel>` must be false.

The same holds true for `three_way_comparable_with`.

4 Proposed wording

In [concepts.lang], the following exposition-only concept is added, intended to detect that there exists a common supertype of `T` and `U` as described earlier:

> **Common supertypes** **[concept.commonsupertype]**
>
> For two types `T` and `U`, if `common_reference_t<const remove_cvref_t<T>&, const remove_cvref_t<U>&>` is well-formed and denotes a type `C` such that both `convertible_to<T&&, const C&>` and `convertible_to<U&&, const C&>` are modeled, then `T` and `U` share a *common comparison supertype* `C`.

```
template<class T, class U>
  concept common-comparison-supertype-with = // exposition only
    same_as<
      common_reference_t<
        const remove_cvref_t<T>&,
        const remove_cvref_t<U>&>,
      common_reference_t<
        const remove_cvref_t<U>&,
        const remove_cvref_t<T>&>> &&
    convertible_to<T&&,
      const common_reference_t<
        const remove_cvref_t<T>&,
        const remove_cvref_t<U>&>&> &&
    convertible_to<U&&,
      const common_reference_t<
        const remove_cvref_t<T>&,
        const remove_cvref_t<U>&>&>;
```

Let `C` be `common_reference_t<const T&, const U&>`. Let `t1` and `t2` be equality-preserving expressions such that `decltype((t1))` and `decltype((t2))` are each `remove_cvref_t<T>`, and let `u1` and `u2` be equality-preserving expressions such that `decltype((u1))` and `decltype((u2))` are each `remove_cvref_t<U>`. `T` and `U` model *common-comparison-supertype-with*`<T, U>` only if:

- `static_cast<const C&>(move(t1))` equals `static_cast<const C&>(move(t2))` if and only if `t1` equals `t2`, and
- `static_cast<const C&>(move(u1))` equals `static_cast<const C&>(move(u2))` if and only if `u1` equals `u2`.

In [cmp.concept]:

```
template<class T, class U, class Cat = partial_ordering>
  concept three_way_comparable_with =
    three_way_comparable<T, Cat> &&
    three_way_comparable<U, Cat> &&
    common_reference_with<
      const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    common-comparison-supertype-with<T, U> &&
    three_way_comparable<
      common_reference_t<
        const remove_reference_t<T>&, const remove_reference_t<U>&>, Cat> &&
    weakly-equality-comparable-with<T, U> &&
    partially-ordered-with<T, U> &&
    requires(const remove_reference_t<T>& t, const remove_reference_t<U>& u) {
      { t <=> u } -> compares-as<Cat>;
      { u <=> t } -> compares-as<Cat>;
    };
```

~~Let `t` and `u` be lvalues of types `const remove_reference_t<T>` and `const remove_reference_t<U>`, respectively.~~ Let `C` be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`. `T`, `U`, and `Cat` model `three_way_comparable_with<T, U, Cat>` only if given lvalues `t` and `u` of types `remove_cvref_t<T>` and `remove_cvref_t<U>`, respectively:

- `t <=> u` and `u <=> t` have the same domain,
- `((t <=> u) <=> 0)` and `(0 <=> (u <=> t))` are equal,
- `(t <=> u == 0) == bool(t == u)` is `true`,

— `(t <=> u != 0) == bool(t != u)` is `true`,

— ~~`Cat(t <=> u) == Cat(C(t) <=> C(u))` is `true`,~~

— The following fragment returns `true`:

```
const auto cat = Cat(t <=> u);
return cat == Cat(
  static_cast<const C&>(move(t))
  <=> static_cast<const C&>(move(u)));
```

— `(t <=> u < 0) == bool(t < u)` is `true`,

— `(t <=> u > 0) == bool(t > u)` is `true`,

— `(t <=> u <= 0) == bool(t <= u)` is `true`,

— `(t <=> u >= 0) == bool(t >= u)` is `true`, and

— if `Cat` is convertible to `strong_ordering`, `T` and `U` model `totally_ordered_with<T, U>`.

In [concept.equalitycomparable]:

Concept `equality_comparable` [concept.equalitycomparable]

```
template<class T, class U>
  concept equality_comparable_with =
    equality_comparable<T> && equality_comparable<U> &&
    ~~common_reference_with<~~
      ~~const remove_reference_t<T>&,~~
      ~~const remove_reference_t<U>&> &&~~
    common-comparison-supertype-with<T, U> &&
    equality_comparable<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    weakly-equality-comparable-with<T, U>;
```

Given types `T` and `U`, let ~~`t` be an lvalue of type `const remove_reference_t<T>`, `u` be an lvalue of type `const remove_reference_t<U>`, and~~ `C` be:

```
common_reference_t<
  const remove_reference_t<T>&,
  const remove_reference_t<U>&>
```

`T` and `U` model `equality_comparable_with<T, U>` only if `bool(t == u) == bool(C(t) == C(u))`.

`T` and `U` model `equality_comparable_with<T, U>` only if given lvalues `t` and `u` of types `remove_cvref_t<T>` and `remove_cvref_t<U>`, respectively, the following fragment returns true:

```
const bool eq = bool(t == u);
return eq == bool(
  static_cast<const C&>(move(t))
  == static_cast<const C&>(move(u)));
```

In [concept.totallyordered]:

```
template<class T, class U>
  concept totally_ordered_with =
    totally_ordered<T> && totally_ordered<U> &&
    equality_comparable_with<T, U> &&
    totally_ordered<
      common_reference_t<
        const remove_reference_t<T>&,
        const remove_reference_t<U>&>> &&
    partially-ordered-with<T, U>;
```

Given types `T` and `U`, let ~~`t` be an lvalue of type `const remove_reference_t<T>`, `u` be an lvalue of type `const remove_reference_t<U>`, and~~ `C` be:

```
common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>
```

`T` and `U` model `totally_ordered_with<T, U>` only if given lvalues `t` and `u` of types `remove_cvref_t<T>` and `remove_cvref_t<U>`, respectively,, the following fragments return true:

- ~~`bool(t < u) == bool(C(t) < C(u))`.~~
- ~~`bool(t > u) == bool(C(t) > C(u))`.~~
- ~~`bool(t <= u) == bool(C(t) <= C(u))`.~~
- ~~`bool(t >= u) == bool(C(t) >= C(u))`.~~
- ~~`bool(u < t) == bool(C(u) < C(t))`.~~
- ~~`bool(u > t) == bool(C(u) > C(t))`.~~
- ~~`bool(u <= t) == bool(C(u) <= C(t))`.~~
- ~~`bool(u >= t) == bool(C(u) >= C(t))`.~~

- ```
  const bool r = bool(t < u);
  return r == bool(
    static_cast<const C&>(move(t))
    < static_cast<const C&>(move(u)));
  ```
- ```
  const bool r = bool(t > u);
  return r == bool(
    static_cast<const C&>(move(t))
    > static_cast<const C&>(move(u)));
  ```
- ```
  const bool r = bool(t <= u);
  return r == bool(
    static_cast<const C&>(move(t))
    <= static_cast<const C&>(move(u)));
  ```
- ```
  const bool r = bool(t >= u);
  return r == bool(
    static_cast<const C&>(move(t))
    >= static_cast<const C&>(move(u)));
  ```
- ```
  const bool r = bool(u < t);
  return r == bool(
    static_cast<const C&>(move(t))
    < static_cast<const C&>(move(u)));
  ```
- ```
  const bool r = bool(u > t);
  return r == bool(
    static_cast<const C&>(move(t))
    > static_cast<const C&>(move(u)));
  ```

— `const bool r = bool(u <= t);`
`return r == bool(`
`static_cast<const C&>(move(t))`
`<= static_cast<const C&>(move(u)));`

— `const bool r = bool(u >= t);`
`return r == bool(`
`static_cast<const C&>(move(t))`
`>= static_cast<const C&>(move(u)));`

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

Document history

— **R0**, 2021-07-10 : Initial version.

Acknowledgements

Many thanks to:

— Matthew Rodusek for their question on Stack Overflow which brought this issue to my attention.

— Tim Song for helping me gain a mathematical understanding of cross-type equality.

References

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

[Stroustrup2012] Bjarne Stroustrup and Andrew Sutton. A Concept Design for the STL. https://wg21.link/n3351, 2012 (accessed 2021-06-30).