Move-only types for equality_comparable_with, totally_ordered_with, and three_way_comparable_with

Document #:	P2404R2
Date:	2022-01-19
Project:	Programming Language C++
Audience:	LEWG and SG9

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

Abstract

None of `equality_comparable_with`, `totally_ordered_with`, or `three_way_comparable_with` support move-only types. For move-only types, these concept's common reference requirement currently ends up requiring that the two types `const T&` and `const U&` can be converted to the non-reference `common_reference_t`, meaning that it requires `T` and `U` to be copyable. This common reference requirement should be relaxed to support these move-only types, effectively turning the common reference requirement into a common *supertype* requirement, as the original reason to require formable references no longer exists.

Contents

Document history

— **R2**, 2022-01-19: Changes from R1:

  — Added a feature test macro. See 3.7. This feature test macro is also in the proposed wording.

  - Factor out the `remove_cvref_t` from the proposed wording for the exposition-only concept. It was quite difficult to read when every `T` and `U` was instead `remove_cvref_t<T>` and `remove_cvref_t<U>`; factoring it out solves the readability issue. `common_reference_t<const T&, const U&>` was also factored out. Note that the intent (5) was also modified to clarify these changes. Note that the implementation experience (4) was *not* updated to verify that this factoring out still passes the tests as stated there.

— **R1**, 2021-12-21: Changes from R0:

  - Rebased on top of N4901.
  - Replaced disjunction with atomic constraint. See 3.1.1.
  - Fixed a bug in the wording of *`common-comparison-supertype-with`*; each `T` and `U` should have `remove_cvref_t`. Section 3.2 already discussed this, but it was missed when applying it to the wording.
  - Add a breakage example and reorganize breakages. See 3.4.1 for the additional example. 3.4.2 is a reorganization of the prior breakage examples.
  - Address monomorphic functions potentially requiring runtime common type conversion. See 3.4.3.
  - Removed a discussion of a partial solution, as there was no interest in it and because such a partial solution comes with all of the drawbacks of the full solution.
  - Expand the discussion of why this paper does not propose to entirely remove the common reference requirements or strip it down to *`weakly-equality-comparable-with`*. See 3.6.
  - Expand implementation experience. See 4.1 for the new section; 4.2 is the same as R0.
  - Minor rephrasing and reformatting in the document.

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

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 they have the same common reference requirement, this also applies to `three_way_comparable_with` and `totally_ordered_with`.

1.2 Specific Code Changes

Some specific examples of code which this paper will simplify can be found in Table 1, given:

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

  // Move-only
  bigint(const bigint&) = delete;
  bigint(bigint&&) noexcept = default;
  bigint& operator=(const bigint&)
    = delete;
  bigint& operator=(bigint&&) noexcept
    = default;

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

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

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

  strong_ordering operator<=>(
    const copyable_bigint&) const;
  bool operator==(
    const copyable_bigint&) const;

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

Table 1: Examples which are simplified by this paper

Before

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

After

```
auto remove_zeros(
  vector<bigint>& range)
{
  return ranges::remove(range, 0);
}
```

Before

```
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(), x);
}
```

After

```
auto find_sorted(
  vector<bigint>& range, int x)
{
  return ranges::lower_bound(range, x);
}
```

Before

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

After

```
bool is_same(
  const vector<bigint>& lhs,
  const vector<copyable_bigint>& rhs)
{
  return ranges::equal(lhs, rhs);
}
```

Before

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

After

```
bool multiset_includes(
  const vector<bigint>& lhs,
  const vector<copyable_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[1], 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 describes this conundrum [Stroustrup2012, 16]. 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`, 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>>;
```

[N4901, 548]

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` and `totally_ordered_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`, but `totally_ordered_with` receives this requirement transitively through `equality_comparable_with`.

[1] Except for implementation details, the common reference conversion will never happen at runtime. See 3.4.3 for a more in-depth discussion.

3 Design

3.1 Overview

The problem with the *comparison_relation*_with concepts is 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 *comparison_relation*_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) comparison functions. This means that it is okay to make it require extreme acrobatics or even make it impossible to write a bool equal_by_common_reference(T, U) function, and similarly for the other comparison relations.

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 require copying the T but instead look for any valid conversion, which can be accomplished by using convertible_to<const T&, C> || convertible_to<T&&, C> (and similarly for U).

Taking both solutions together yields convertible_to<const T&, const C&> || convertible_to<T&&, const C&>, and this combined solution does not invalidate any of the prior arguments.

3.1.1 Avoiding disjunctions

Disjunctions in concepts may slow down compilation times due to added costs in the conversion to disjunctive normal form, so it is desirable to avoid them. As such, this new disjunction should be made into an atomic constraint to avoid this issue. Lost functionality is not a concern in this case, because users are not expected to find it useful to have subsumption between equality_comparable_with<T, U> and convertible_to<const T&, const common_reference_t<const T&, const U&>&> or the other similar cases.

Avoiding the disjunctions in this case is easily resolved by forcing the disjunction to be an atomic constraint via a requires expression and nested requirements:

```
requires {
  requires convertible_to<const T&, const C&> || convertible_to<T&&, const C&>;
}
```

In this particular case, we have two disjunctions, one for T and one for U. Having a conjunction between atomic constraints is not useful, so the proposed wording will merge them into a single atomic constraint.

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 and with its usage in other contexts. 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 need to account for that by stripping the cvref-qualifiers. `const` and references are mathematically meaningless, so stripping the cvref-qualifiers does not cause any issues 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 will copy or move to a `const&` by modifying the wording to use both `static_cast<const C&>(t)` and `static_cast<const C&>(move(t))` to allow for either the copying constructor or the moving constructor to be used, whichever is valid.

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 allowed to be 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

3.4.1 Breakage of existing code

Changing any existing concept is a breaking change for many reasons. In particular, this change breaks anyone relying on the *`comparison_relation`*`_with` concepts implying `common_reference_with`. This is by design; removing this requirement is exactly the intention of the paper.

Figure 1 shows such an example. Currently, the code in Figure 1 is allowed. When using `equality_comparable_with<T, U>`, the implementer was free to expect to be able to convert to the common reference as an implementation detail. This expectation is broken by this paper. To fix the code, either the implementer must remove the implementation detail by directly using `*first1 == *first2`, or if they wish to keep the conversion as an implementation detail, the implementer would have to add the requirement `common_reference_with<const T&, const U&>`. Note that this is

Figure 1: Example of code broken due to relying on `common_reference_with` as an implementation detail.

```
template <ranges::range R1, ranges::range R2>
  requires equality_comparable_with<
    ranges::range_reference_t<R1>,
    ranges::range_reference_t<R2>>
int count_equals(R1&& r1, R2&& r2) {
  using C = common_reference_t<
    ranges::range_reference_t<R1>,
    ranges::range_reference_t<R2>>;

  auto first1 = ranges::begin(r1);
  auto first2 = ranges::begin(r2);
  const auto last1 = ranges::end(r1);
  const auto last2 = ranges::end(r2);

  int count = 0;

  while (first1 != last1 && first2 != last2) {
    // Copies to the common reference.
    // This is allowed before this paper, but not after.
    C c1 = *first1;
    C c2 = *first2;

    if (c1 == c2) ++count;

    ++first1;
    ++first2;
  }

  return count;
}
```

enough, because although this paper removes `common_reference_with` from its definition, the way the paper specifies `equality_comparable_with` means that `common_reference_with` is applying a stricter condition on the same `common_reference_t`.

That said, although Figure 1 is broken by this paper, because this paper is a relaxation, `count_equals` will continue to work as before for any types which it accepted before. The only types which fail to be converted to the common reference type are those which previously did not pass the `equality_comparable_with` requirement. In other words, this paper does break Figure 1 and other algorithms relying on `common_reference_with` as an implementation detail, but only as it pertains to whether the algorithm is appropriately constrained; any existing code using the algorithm would continue to compile.

3.4.2 Breakage in existing code: edge cases

Figure 2 shows more examples of code broken due to this paper, but these cases are considered unlikely to occur. Figure 2a demonstrates that behavior can change for types which only now meet `equality_comparable_with` by changing which overload is chosen for an overload set for when passed these types. Figure 2b demonstrates that the removed subsumption with `common_reference_with` can break code which was relying on that subsumption. Figure 2c shows that code which deliberately relies on the result of the changed concepts can be broken due to the change in its value.

Figure 2: Examples of code broken due to edge cases in this paper's changes.

(a) Behavior change of a fallback against the modified concept's.

```
template <typename T, typename U>
struct equality_traits;

// Assume bigint and
// copyable_bigint are as before.
template <>
struct equality_traits<
  bigint, copyable_bigint> {
  // A manual implementation which, for
  // some reason, does not use operator==.
  static bool equals(
    const bigint&, const copyable_bigint&);
};

template <typename T, typename U>
  requires equality_comparable_with<T, U>
bool fancy_equals(const T& t, const U& u) {
  return t == u;
}

template <typename T, typename U>
bool fancy_equals(const T& t, const U& u) {
  return equality_traits<T, U>::equals(t, u);
}

// Calling code
bigint a = ...;
copyable_bigint b = ...;
```

(b) Removed subsumption makes the overload ambiguous.

```
// Some type using a different spelling of equality.
class fancy_int {
  int x;

public:
  fancy_int(int x) : x(x) {}

  bool equals(int y) const { return x == y; }
};

template<class T, class U>
  requires equality_comparable_with<T, U>
bool attempted_equals(const T& t, const U& u) {
  return t == u;
}

template<class T, class U>
  requires common_reference_with<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&>
bool attempted_equals(const T& t, const U& u) {
  static_assert(requires { { t.equals(u) }
    -> convertible_to<bool>; });
  return t.equals(u);
}

auto test1(const shared_ptr<int>& p) {
  return attempted_equals(p, nullptr);
  // With this proposed change:
  // error: call of overloaded 'common()' is ambiguous
}

auto test2(const fancy_int& x, int y) {
  // Still works:
  return attempted_equals(x, y);
}
```

(c) Directly testing the concept's value.

```
template <typename T>
void questionable(unique_ptr<T> p) {
  if constexpr (equality_comparable_with<
      unique_ptr<T>, nullptr_t>) {
    1 / 0; // Cause undefined behavior.
  }
}
```

Although these examples are broken by this change, the drawbacks of these breakages are low compared to the benefits of enabling move-only types for the *comparison_relation*_with concepts. For Figure 2a, despite the change in behavior by which function is called, either the end result will be the same or the code already had a bug where the semantic meaning of "equals" was not respected by `equality_traits<bigint, copyable_bigint>::equals(...)`. Despite Figure 2b showing subsumption being broken, the loss of subsumption generally results in hard errors rather than silently incorrect behavior changes, and subsumption can always be regained by adding the additional constraint of `common_reference_with<const T&, const U&>`. Figure 2c is pathological, as the code modifies semantics or even creates undefined based on type introspection whose answer may change. Refusing to break such pathological code is to forbid changing the standard, as adding member functions, overloads, and so on also breaks similar code.

3.4.3 Why do we never need a runtime common supertype conversion?

Although the current specification of `equality_comparable_with` allows implementations to convert the arguments to the common supertype at runtime, this never needs to happen and can instead be replaced with a direct comparison of the two types. This specifically applies for algorithms which want to use `operator==` rather than an arbitrary predicate, where the predicate is not constrained with `equality_comparable_with` like `ranges::equal_to` is. For arbitrary predicates in the algorithms, there are cases where the equivalence relation required of the predicate can perform a conversion to the common supertype if the user specifies it. Such a case is presented here, along with why this does not affect this paper's change.

In *Iterators++, Part 3*, Eric Niebler describes how the intersection of proxy iterators with `std::ranges::unique_copy` produces a case where users may wish to use a conversion to `common_reference_t` in a predicate in order to use a monomorphic function rather than a template [Niebler2015]. To summarize the point in the article, code such as the following fails to compile (example modified from [Niebler2015]):

```
vector<bool> vec = some_initial_value();
using R = vector<bool>::reference;
ranges::unique_copy(vec,
  ostream_iterator<bool>{cout, " "},
  [](R b1, R b2) { return b1 == b2; });
```

But using the common reference type allows a monomorphic predicate to compile (again, example modified from [Niebler2015]):

```
vector<bool> vec = some_initial_value();
using C = iter_common_reference_t<vector<bool>::iterator>;
ranges::unique_copy(vec,
  ostream_iterator<bool>{cout, " "},
  [](C b1, C b2) { return b1 == b2; });
```

However, this example does not apply to `equality_comparable_with` since the user is specifying a predicate rather than relying on `ranges::equal_to`. That is to say, because the user specifies their own predicate, there is no `equality_comparable_with` requirement in Niebler's example, only a requirement that the predicate forms an equivalence relation. Even if the user added an `equality_comparable_with<C, C>` requirement, there is no issue, because the equality happens after the conversion to the common reference.

This paper only affects similar user provided predicates if the user provided a templated predicate using `equality_comparable_with` which delegated to a monomorphic common supertype predicate without also specifying `common_reference_with`; a breaking change already mentioned in 3.4.1 and Figure 1.

Figure 3: An example demonstrating a class deleting the constructor from an rvalue reference.

```
// Assume bigint is as before.
class bigint_cref {
public:
  bigint_cref(const bigint&);
  // Forbid construction from rvalue references:
  bigint_cref(const bigint&&) = delete;

  strong_ordering operator<=>(bigint_cref) const;
  bool operator==(bigint_cref) const;

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

static_assert(equality_comparable_with<bigint, bigint_cref>);
```

3.5 Why is `convertible_to<T&&, const C&>` insufficient?

It may appear that we could simplify `convertible_to<const T&, const C&> || convertible_to<T&&, const C&>` to just `convertible_to<T&&, const C&>`, as a constructor that takes a `const T&` can also always take a `T&&`. However, this forgets the case of deleted rvalue overloads as in Figure 3.

Figure 3 compiles fine before this paper and with the disjunction. However, with only `convertible_to<T&&, const C&>` and no disjunction, the `static_assert` fails, with the compilation error including the note: "'`convertible_to<bigint &&, const bigint_cref &>`' evaluated to false."

This pattern deletes the rvalue overload of an overload set—the constructor in this case—to attempt to prevent the function from being called with temporaries and solve some lifetime management errors. Although this pattern fails to correctly capture lifetime constraints as rvalue references do not necessarily imply an immediately expiring lifetime, there is currently no way to properly manage lifetime constraints, so this is a pattern that is used not too infrequently. To maintain support of this pattern, this paper uses the disjunction `convertible_to<const T&, const C&> || convertible_to<T&&, const C&>`.

3.6 Could we remove the common reference requirement or move to weak equality?

A common suggestion has been to remove the common reference requirement altogether, leaving us with approximately *`weakly-equality-comparable-with`* and *`partially-ordered-with`*, possibly with additional semantic requirements. This paper deliberately does not propose such a change for two reasons:

1. Stripping the common reference requirement is orthogonal to the changes presented in this paper. The changes that this paper proposes do not in any way prevent such a change from being made in the future.

2. It is the author's belief that the common reference requirements are necessary for these concepts to be sound. The common reference requirements exist to provide at least some verification that the types meet the mathematical models of equality and ordering. The common reference requirements very closely match the mathematical model for extending equality and ordering to be heterogeneous (see 2.2). It seems likely that removing these requirements would easily allow types through which do not meet the mathematical models, which would lead to serious repercussions on the ability to reason about code. It seems unwise

to ignore the analysis in the Palo Alto report [Stroustrup2012] which explains that these requirements are important. See 2.2 for more information on the Palo Alto report's discussion of these requirements.

It is important to note that a large number of types—including in the standard library—use `operator==` for something other than equality. Removing the common reference requirements erroneously allows these types to meet `equality_comparable_with`. It turns out to be easy to write an `operator==(T, U)` which feels like 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.

As an example, iterators and sentinels have a cross-type `operator==(iterator, sentinel)` which feels like equality and could form an equivalence relation, 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, because the relevant equality operators are not consistent with each other.

The same holds true for `three_way_comparable_with` and `totally_ordered_with`.

This paper intentionally only attempts to appropriately expand the concepts under the assumption that the current model is correct. If it is possible to strip the requirements down to *`weakly-equality-comparable-with`* and *`partially-ordered-with`* while retaining a sound model, such a significant change—involving reworking what it means for a type to be *regular* and working out all of the mathematical implications—is out of scope for this paper.

3.7 Feature Test Macro

A feature test macro should be added. Two possible approaches could make sense: either update `__cpp_lib_concepts` to a higher value or add a new feature test macro. This paper proposes to add a new feature test macro: `__cpp_lib_relaxed_comparison_concepts`. This feature test macro should be available in `<version>` as well as the headers which provide any of the three concepts `equality_comparable_with`, `totally_ordered_with`, or `three_way_comparable_with`, so additionally `<concepts>` and `<compare>`.

4 Testing the proposed implementation

4.1 Full implementation

The proposed wording (6) was implemented in libc++ [Libcxx] and libstdc++ [Libstdcxx]. Each standard library was then built with GCC 11.1 [GCC] and the full test suites were run. Clang++ 14.0.0 [Clang] was also used to build and test libc++.

Note that these changes were not implemented in the Microsoft STL [MicrosoftSTL] due to lack of access to a Windows machine.

To summarize the results, the proposed changes do not have any further repercussions in broken tests in libc++ nor in libstdc++.

4.1.1 Libc++

Libc++ was modified at commit `d5b73a70a0611fc6c082e20acb6ce056980c8323` to incorporate the proposed changes.

Clang++ 14.0.0 successfully built libc++. There were 6 failed tests in the test suite, all of which were tests on the constrained comparison function objects—such as `ranges::equal_to`—to verify that they cannot be invoked with a move-only object. As that is precisely the change which this paper attempts to fix, these failures are expected.

GCC 11.1 successfully built libc++. There were 6 failed tests in the test suite, precisely those 6 tests which failed for Clang++ 14.0.0. As such, these failures are expected.

Thus we can confirm that nothing in libc++'s test suite is broken due to this paper's change.

4.1.2 Libstdc++

Libstdc++ was modified at commit `df1a0d526e2e4c75311345c0b73ce8483e243899` to incorporate the proposed changes.

GCC 11.1 successfully built libstdc++. A single test failed at runtime, `30_threads/jthread/95989.cc`, which failure happened without this change and is also not related to this change. All other tests passed.

Clang++ 14.0.0 was not tested due to lack of time.

4.2 Running concept-specific tests

Rather than changing the standard libraries, this experiment implemented the new concepts separately, then ran all of the test cases referencing the changed concepts in the standard libraries' test suites over the alternative implementations. In contrast with 4.1, this allowed testing the Microsoft STL test suite, the Microsoft compiler, and how the different implementations perform on each other's test suites.

These concepts' tests were gathered from the libc++ [Libcxx] test suite and the Microsoft STL [MicrosoftSTL] test suite at commits `1c69005c2e11414669ac8ba094a9b059920936db` and `280347a4309eaaf5f1bba3b1ad98a27687b9d9c3` respectively. At the time of writing, libstdc++ [Libstdcxx] at commit `a7098d6ef4e4e799dab8ef925c62b199d707694b` did not have tests specifically for these concepts. These gathered tests were then modified to be run over the concepts from the proposed wording (6), thus testing the proposed concepts against both libc++'s and the Microsoft STL's tests.

With the proposed changes, all the tests pass for all three of `equality_comparable_with`, `totally_ordered_with`, and `three_way_comparable_with` except tests which fail even without these changes due to compiler bugs or incomplete implementations. That is to say, the only tests that fail do so for unrelated reasons. To summarize the test results:

- A single test fails for GCC 11.1, as it claims that `nullptr_t` meets `totally_ordered`. This is because GCC 11.1 has relational operators defined for `nullptr_t`. This test failure is unrelated to the proposed changes.
- Two tests fail for MSVC 19.29.30130.2:
  - MSVC does not support `static_assert(requires { ... })`, so it fails to parse a test in that form. This test failure is unrelated to the proposed changes.
  - MSVC claims `!equality_comparable_with<nullptr_t, int (&)()>`, but libc++ includes such a test in its test suite. This test failure is unrelated to the proposed changes.
- All tests pass for Clang 12.0.0.

In short, the proposed changes do not break any of these concepts' tests in libc++ or the Microsoft STL.

5 Intent

To summarize the intent of the proposed changes, given `C = common_reference_t<const T&, const U&>`, this paper intends to relax the common reference requirements by:

— Relaxing the `convertible_to<const T&, C>` invocations to allow types satisfying `same_as<remove_cvref_t<T>, remove_cvref_t<C>>` to meet the concept without requiring copying the `T`.

— Relaxing the `convertible_to<const T&, C>` invocations to allow for types where it is possible to convert a `T` to `C`, but only via moving the `T`. Recall that the move does not happen at runtime, so despite allowing moves, we are not changing any values (3.1).

The following proposed wording (6) uses some patterns whose intent is as follows:

— *`COMMON`*`(...)` is intended to convert the `...` to the common reference via copying or moving the value, whichever is valid. This should allow for types which can be moved to the common reference, but not copied to the common reference.

— *`COMMON`*`(...)` uses `static_cast<const C&>(...)` in its conversions, but this is intended solely to convert to a `const C&` instead of `C` directly. This is not intended to require explicit conversions to be taken, which should already be forbidden by the fact that the syntactic requirements require implicit conversions via `convertible_to`.

— Each expression which previously had conversions to the common type is split into two pieces, first evaluating without the conversion, then comparing this prior evaluation against the result after the conversion. This is intended to avoid any issue where moving the `T` or `U` lvalues via *`COMMON`*`(...)` changes the value of the objects before we perform the heterogeneous evaluation.

— The original semantic requirements used lvalues of type `const remove_reference_t<T>` and similarly for `U`, but these lvalues were changed to be of type `remove_cvref_t<T>` and `remove_cvref_t<U>`. This change is not intended to say that the concepts only work with non-const lvalues, but it is instead intended to allow *`COMMON`*`(...)` to properly move if necessary by creating `T&&` and `U&&` instead of `const T&&` and `const U&&`.

— In the wording for *`common-comparison-supertype-with`*, a helper concept is used, with the name of *`common-comparison-supertype-with-impl`*. This is solely to improve the readability of the concept definition, by factoring out the `remove_cvref_t<T>` and `remove_cvref_t<U>` and also by factoring out the `common_reference_t<const T&, const U&>`. This is intended to be exactly the same as if everything was textually inlined into the definition of *`common-comparison-supertype-with`*, meaning that any potential change in semantics between the two is not intended. In particular, the default argument for factoring out the `common_reference_t` should not block proper syntactic checking, which already works because the context where the possibly ill formed `common_reference_t<const T&, const U&>` appears is in the definition of *`common-comparison-supertype-with`*.

6 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<const T&, const C&> || convertible_to<T&&, const C&>` and `convertible_to<const U&, const C&> || convertible_to<U&&, const C&>` are modeled, then `T` and `U` share a *common comparison supertype* `C`.

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

      requires convertible_to<const U&, const C&> ||
        convertible_to<U&&, const C&>;
    };

template<class T, class U>
  concept common-comparison-supertype-with = // exposition only
    common-comparison-supertype-with-impl<
      remove_cvref_t<T>, 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>`. Let *COMMON*`(...)` be `static_cast<const C&>(...)` if `static_cast<const C&>(...)` is a valid expression and `static_cast<const C&>(move(...))` otherwise. `T` and `U` model *common-comparison-supertype-with*`<T, U>` only if:

— *COMMON*`(t1)` equals *COMMON*`(t2)` if and only if `t1` equals `t2`, and

— *COMMON*`(u1)` equals *COMMON*`(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>&>`. Let *COMMON*`(...)` be `static_cast<const C&>(...)` if `static_cast<const C&>(...)` is a valid expression and `static_cast<const C&>(move(...))` otherwise. `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))`~~
After evaluating `const auto cat = Cat(t <=> u);`,
`cat == Cat(`*COMMON*`(t) <=> `*COMMON*`(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`,

— `(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))`. Let *COMMON*`(...)` be `static_cast<const C&>(...)` if `static_cast<const C&>(...)` is a valid expression and `static_cast<const C&>(move(...))` otherwise. `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, after evaluating `const bool eq = bool(t == u);`, `eq == bool(`*COMMON*`(t) == `*COMMON*`(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>&>
```

Let *`COMMON`*`(...)` be `static_cast<const C&>(...)` if `static_cast<const C&>(...)` is a valid expression and `static_cast<const C&>(move(...))` otherwise. `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:

- ~~`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))`.~~

- After evaluating `const bool r = bool(t < u);`, `r == bool(`*`COMMON`*`(t) < `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(t > u);`, `r == bool(`*`COMMON`*`(t) > `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(t <= u);`, `r == bool(`*`COMMON`*`(t) <= `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(t >= u);`, `r == bool(`*`COMMON`*`(t) >= `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(u < t);`, `r == bool(`*`COMMON`*`(t) < `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(u > t);`, `r == bool(`*`COMMON`*`(t) > `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(u <= t);`, `r == bool(`*`COMMON`*`(t) <= `*`COMMON`*`(u))` is `true`,
- After evaluating `const bool r = bool(u >= t);`, `r == bool(`*`COMMON`*`(t) >= `*`COMMON`*`(u))` is `true`,

In [version.syn]:

```
#define __cpp_lib_relaxed_comparison_concepts <DATE OF ADOPTION> // also in <concepts>, <compare>
```

The proposed changes are relative to the current working draft [N4901].

Acknowledgements

Many thanks to:

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

— Tim Song for pointing me in the right direction to gain a mathematical understanding of cross-type equality.

— Christopher Di Bella for helping determine whether this paper is mathematically sound.

References

[Clang] https://github.com/llvm/llvm-project/tree/main/clang.

[GCC] git://gcc.gnu.org/git/gcc.git.

[Libcxx] https://github.com/llvm/llvm-project/tree/main/libcxx.

[Libstdcxx] git://gcc.gnu.org/git/gcc.git.

[MicrosoftSTL] https://github.com/microsoft/STL.

[N4901] Thomas Köppe. Working Draft, Standard for Programming Language C++. https://wg21.link/n4901, 2021 (accessed 2021-11-21).

[Niebler2015] Eric Niebler. Iterators++, Part 3. https://ericniebler.com/2015/03/03/iterators-plus-plus-part-3/, 2015 (accessed 2021-11-12).

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