Ranges Design Cleanup
## Contents

1 Abstract
   1.1 Revision History ............................................. 1

2 Deprecate `move_iterator::operator->`
   2.1 Technical Specifications ................................. 1

3 `ref-view => ref_view`
   3.1 Technical Specifications ................................ 2

4 Comparison function object untemplates
   4.1 Technical specifications .................................. 4

5 Reversing a `reverse_view`
   5.1 Technical specifications ................................. 7

6 Use cases left dangling
   6.1 Technical specifications .................................. 7

Bibliography ...................................................... 9
1 Abstract

This paper proposes several small, independent design tweaks to Ranges that came up during LWG review of P0896 “The One Ranges Proposal” ([2]).

All wording sections herein are relative to the post-San Diego working draft.

1.1 Revision History

1.1.1 Revision 1

— Rebase wording onto post-San Diego working draft.
— Strike section that suggested making the exposition-only concepts in [special.mem.concepts] available to users; this part of the proposal did not have consensus in LEWG.
— Update safe_subrange_t to account for potential dangling as well.

1.1.2 Revision 0

— In the beginning, all was cv-void. Suddenly, a proposal emerged from the darkness!

2 Deprecate move_iterator::operator->

C++17 [iterator.requirements.general]/1 states:

... An iterator i for which the expression (*i).m is well-defined supports the expression i->m with the same semantics as (*i).m ...

Input iterators are required to support the -> operator ([input.iterators]), and move_iterator is an input iterator, so move_iterator’s arrow operator must satisfy that requirement, right? Sadly, it does not.

For a move_iterator, *i is an xvalue, so (*i).m is also an xvalue. i->m, however, is an lvalue. Consequently, (*i).m and i->m can produce observably different behaviors as subexpressions - they are not substitutable, as would be expected from a strict reading of “with the same semantics.” The fact that -> cannot be implemented with “the same semantics” for iterators whose reference type is an rvalue was the primary motivation for removing the -> requirement from the Ranges iterator concepts. It would benefit users to deprecate move_iterator’s operator-> in C++20 as an indication that its semantics are not equivalent and that it will ideally go away some day.

2.1 Technical Specifications

— Strike move_iterator::operator-> from the class template synopsis in [move.iterator]:

```cpp
namespace std {
    template<class Iterator>
    class move_iterator {
        [...]
        constexpr iterator_type base() const;
        constexpr reference operator*() const;
        constexpr pointer operator->() const;
    
        constexpr move_iterator& operator++();
        constexpr auto operator++(int);
        [...]
    }
}
```


— Relocate the detailed specification of `move_iterator::operator->` from `move.iter.elem`:

```cpp
constexpr reference operator*() const;
Effects: Equivalent to: return ranges::iter_move(current);
```

```cpp
constexpr pointer operator->() const;
Returns: current.
```

```cpp
constexpr reference operator[](difference_type n) const;
Effects: Equivalent to: ranges::iter_move(current + n);
```

to a new subclause “Deprecated `move_iterator` access” in Annex D between [depr.iterator.primitives] and [depr.util.smartptr.shared.atomic]:

The following member is declared in addition to those members specified in `move.iter.elem`:

```cpp
namespace std {
  template<class Iterator>
  class move_iterator {
  public:
    constexpr pointer operator->() const;
  };
}
```

```cpp
constexpr pointer operator->() const;
Returns: current.
```

3  **ref-view => ref_view**

The authors of P0896 added the exposition-only view type `ref-view` (P0896R4 [range.view.ref]) to serve as the return type of `view::all` ([range.adaptors.all]) when passed an lvalue container. `ref-view` is an “identity view adaptor” – an adaptor which produces a view containing all the elements of the underlying range exactly – of a `Range` of type `T` whose representation consists of a `T*`. A `ref-view` delegates all operations through that pointer to the underlying `Range`.

The LEWG-approved design from P0789R3 “Range Adaptors and Utilities” ([1]) used `subrange<iterator_t<R>, sentinel_t<R>>` as the return type of `view::all(c)` for an lvalue `c` of type `R`. `ref-view` and `subrange` are both identity view adaptors, so this change has little to no impact on the existing design. Why bother then? Despite that replacing `subrange` with `ref-view` in this case falls under as-if, `ref-view` has some advantages.

Firstly, a smaller representation: `ref-view` is a single pointer, whereas `subrange` is an iterator plus a sentinel, and sometimes a size. View compositions store many views produced by `view::all`, and many of those are views of lvalue containers in typical usage.

Second, and more significantly, `ref-view` is future-proof. `ref-view` retains the exact type of the underlying `Range`, whereas `subrange` erases down to the `Range`’s iterator and sentinel type. `ref-view` can therefore easily model any and all concepts that the underlying range models simply by implementing any required expressions via delegating to the actual underlying range, but `subrange` must store somewhere in its representation any properties of the underlying range needed to model a concept which it cannot retrieve from an iterator and sentinel. For example, `subrange` must store a size to model `SizedRange` when the underlying range is sized but does not have an iterator and sentinel that model `SizedSentinel`. If we discover in the future that it is desirable to have the `View` returned by `view::all(container)` model additional concepts, we will likely be blocked by ABI concerns with `subrange` whereas `ref-view` can simply add more member functions and leave its representation unchanged.

We’ve already realized these advantages for view composition by adding `ref-view` as an exposition-only `View` type returned by `view::all`, but users may like to use it as well as a sort of “Ranges reference wrapper”.

3.1 Technical Specifications

— Update references to the name `ref-view` to `ref_view` in `range.adaptors.all`/2:
The name `view::all` denotes a range adaptor object ([range.adaptor.object]). For some subexpression \(E\), the expression `view::all(E)` is expression-equivalent to:

- `DECAY_COPY(E)` if the decayed type of \(E\) models \texttt{View}.
- Otherwise, `ref_view(E)` if that expression is well-formed, where \texttt{ref_view} is the exposition-only \texttt{View} specified below.
- Otherwise, `subrange(E)`.

---

Change the stable name `[range.view.ref]` to `[range.ref.view]` (for consistency with the stable names of the other \texttt{View} classes defined in [range]), retitle to “class template \texttt{ref_view}” and modify as follows:

\[ref\_view\] is a \textit{View} of the elements of some other \texttt{Range}.

```cpp
amespace std::ranges {
template<Range R>
    requires is_object_v<R>
    class \_view\_ref_view : public view_interface<\_view\_ref_view<R>> {
        private:
            R* r_ = nullptr;  // exposition only
        public:
            constexpr \_view\_ref_view() noexcept = default;
            template<not-same-as<\_view\_ref_view> T>
                requires see below
                constexpr \_view\_ref_view(T&& t);
            constexpr R& base() const;
            constexpr iterator_t<R> begin() const { return ranges::begin(*r_); }
            constexpr sentinel_t<R> end() const { return ranges::end(*r_); }
            constexpr bool empty() const
                requires requires { ranges::empty(*r_); }
                { return ranges::empty(*r_); }
            constexpr auto size() const requires SizedRange<R>
                { return ranges::size(*r_); }
            constexpr auto data() const requires ContiguousRange<R>
                { return ranges::data(*r_); }
            friend constexpr iterator_t<R> begin(\_view\_ref_view r)
                { return r.begin(); }
            friend constexpr sentinel_t<R> end(\_view\_ref_view r)
                { return r.end(); }
    };
```  

4 Comparison function object untemplates

[untemp]

During LWG review of P0896’s comparison function objects (P0896R3 [range.comparisons]) we were asked, “Why are we propagating the design of the \texttt{std} comparison function objects, i.e. class templates that you shouldn’t specialize because they cannot be specialized consistently with the \texttt{void} specializations that you actually should be using?” For the Ranges TS, it was a design goal to minimize differences between \texttt{std}
and ranges to ease transition and experimentation. For the Standard, our goal should not be to minimize differences but to produce the best design. (As was evidenced by the LEWG poll in Rapperswil suggesting that we should not be afraid to diverge std and ranges components when there are reasons to do so.)

Absent a good reason to mimic the std comparison function objects exactly, we propose un-\texttt{template}-ing the \texttt{std::ranges} comparison function objects, leaving only concrete classes with the same behavior as the prior \texttt{void} specializations.

### 4.1 Technical specifications

In [functional.syn], modify the declarations of the \texttt{ranges} comparison function objects as follows:

```cpp
namespace ranges {
    // [range.comparisons], comparisons
    template<class T = void>
    requires see below
    struct equal_to;

    template<class T = void>
    requires see below
    struct not_equal_to;

    template<class T = void>
    requires see below
    struct greater;

    template<class T = void>
    requires see below
    struct less;

    template<class T = void>
    requires see below
    struct greater_equal;

    template<class T = void>
    requires see below
    struct less_equal;

    template<> struct equal_to<void>;
    template<> struct not_equal_to<void>;
    template<> struct greater<void>;
    template<> struct less<void>;
    template<> struct greater_equal<void>;
    template<> struct less_equal<void>;
}
```

Update the specifications in [range.comparisons] as well:

2 There is an implementation-defined strict total ordering over all pointer values of a given type. This total ordering is consistent with the partial order imposed by the builtin operators $<$, $>$, $<=$, and $=>$.

```cpp
template<class T = void>
requires EqualityComparable<T> || Same<T, void> || BUILTIN_PTR_CMP
struct equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

3 \texttt{operator()} has effects equivalent to: return \texttt{ranges::equal_to<>}{}(x, y);

```cpp
template<class T = void>
requires EqualityComparable<T> || Same<T, void> || BUILTIN_PTR_CMP
struct not_equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};
```
operator() has effects equivalent to: return !ranges::equal_to<>{}(x, y);

```c++
template<class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct greater {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

operator() has effects equivalent to: return ranges::less<>{}(y, x);

```c++
template<class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct less {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

operator() has effects equivalent to: return ranges::less<>{}(x, y);

```c++
template<class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct greater_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

operator() has effects equivalent to: return !ranges::less<>{}(x, y);

```c++
template<class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

operator() has effects equivalent to: return !ranges::less<>{}(y, x);

```c++
template<> struct equal_to<void> {
  template<class T, class U>
    requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;
  
  using is_transparent = unspecified;
};
```

**Expects:** If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator == comparing pointers of type P, the conversion sequences from both T and U to P shall be equality-preserving ([concepts.equality]).

**Effects:**

- If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator == comparing pointers of type P: returns `false` if either (the converted value of) t precedes u or u precedes t in the implementation-defined strict total order over pointers of type P and otherwise `true`.

- Otherwise, equivalent to: return `std::forward<T>(t) == std::forward<U>(u)`;

```c++
template<> struct not_equal_to<void> {
  template<class T, class U>
    requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;
  
  using is_transparent = unspecified;
};
```

operator() has effects equivalent to:

```c++
return !ranges::equal_to<>{}(std::forward<T>(t), std::forward<U>(u));
```
template<> struct greater<void> {
    template<class T, class U>
        requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
    constexpr bool operator()(T& t, U& u) const;
    using is_transparent = unspecified;
};
12
operator() has effects equivalent to:
    return ranges::less<>(std::forward<U>(u), std::forward<T>(t));

operator() has effects equivalent to:
    return ranges::less<>(std::forward<T>(t), std::forward<U>(u));

template<> struct less<void> {
    template<class T, class U>
        requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
    constexpr bool operator()(T&& t, U&& u) const;
    using is_transparent = unspecified;
};
13
Expects: If the expression std::forward<T>(t) < std::forward<U>(u) results in a call to a built-in operator < comparing pointers of type P, the conversion sequences from both T and U to P shall be equality-preserving ([concepts.equality]). For any expressions ET and EU such that decltype((ET)) is T and decltype((EU)) is U, exactly one of ranges::less<>(ET, EU), ranges::less<>(EU, ET), or ranges::equal_to<>(ET, EU) shall be true.

Effects:
14
— If the expression std::forward<T>(t) < std::forward<U>(u) results in a call to a built-in operator < comparing pointers of type P; returns true if (the converted value of) t precedes u in the implementation-defined strict total order over pointers of type P and otherwise false.
14.1
— Otherwise, equivalent to: return std::forward<T>(t) < std::forward<U>(u);
14.2

operator() has effects equivalent to:
    return !ranges::less<>(std::forward<T>(t), std::forward<U>(u));

template<> struct greater_equal<void> {
    template<class T, class U>
        requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
    constexpr bool operator()(T& t, U& u) const;
    using is_transparent = unspecified;
};
15
operator() has effects equivalent to:
    return !ranges::less<>(std::forward<T>(t), std::forward<U>(u));

operator() has effects equivalent to:
    return !ranges::less<>(std::forward<U>(u), std::forward<T>(t));

Strip <> from occurrences of ranges::equal_to<, ranges::less<, etc. in: [defs.projection], [iterator.synopsis], [commonalgoreq.general]/2, [commonalgoreq.mergeable], [commonalgoreq.sortable], [range.syn], [range.adaptors.split_view], [algorithm.syn], [alg.find], [alg.find.end], [alg.find.first.of], [alg.adjacent.find], [alg.count], [alg.mismatch], [alg.equal], [alg.is_permutation], [alg.search], [alg.replace], [alg.remove], [alg.unique], [sort], [stable.sort], [partial.sort], [partial.sort.copy], [is.sorted], [alg.mth.element], [lower.bound], [upper.bound], [equal.range], [binary.search], [alg.merge], [includes], [set.union], [set.intersection], [set.difference], [set.symmetric.difference], [push.heap], [pop.heap], [make.heap], [sort.heap], [is.heap], [alg.min.max], [alg.lex.comparison],
and \[\text{alg.permutation.generators}\].

5 Reversing a reverse\_view \[\text{weiv\_esrever}\]

\text{view::reverse} in P0896 is a range adaptor that produces a reverse\_view which presents the elements of the underlying range in reverse order - from back to front. reverse\_view does so via the expedient mechanism of adapting the underlying view’s iterators with \text{std::reverse\_iterator}. Reversing a reverse\_view produces a view of the elements of the original range in their original order. While this behavior is correct, it is likely to exhibit poor performance.

We propose that the effect of \text{view::reverse(r)} when \(r\) is an instance of reverse\_view should be to simply return the underlying view directly. This behavior is both simple to specify and efficient to implement.

Similarly, reversing a subrange whose iterator and sentinel are reverse\_iterators can be made more efficient by yielding a subrange of 'unwrapped' iterators. Note that in this case we should take care to preserve any stored size information in the original subrange, since the size of the unwrapped base range is the same.

5.1 Technical specifications \[\text{sdrow.weiv\_esrever}\]

— Modify the specification of \text{view::reverse} in \[\text{range.reverse.adaptor}\] as follows:

\[
\text{view::reverse} \quad \text{denotes a range adaptor object (\text{[range.adaptor.object]})}. \text{For some subexpression } E, \text{the expression } \text{view::reverse}(E) \text{ is expression-equivalent to } \text{reverse_view}(E).
\]

— If the type of \(E\) is a cv-qualified specialization of \text{reverse_view}, \(E\).base().

— Otherwise, if the type of \(E\) is cv-qualified

\[
\text{subrange<reverse\_iterator}<I>, \text{reverse\_iterator}<I>, K>
\]

for some iterator type \(I\) and value \(K\) of \text{subrange\_kind},

\[
\text{subrange}<I, I, K>(E.end().base(), E.begin().base(), E.size())
\]

if \(K\) is \text{subrange\_kind::sized} and

\[
\text{subrange}<I, I, K>(E.end().base(), E.begin().base())
\]

otherwise, except that in either case \(E\) is evaluated only once.

— Otherwise, \text{reverse\_view}(E).

6 Use cases left dangling \[\text{dangle}\]

What does this program fragment do in P0896?

```cpp
std::vector<int> f();
std::ranges::copy(f(), o).out;
```

how about this one:

```cpp
std::ranges::copy(f(), std::ostream_iterator<int>{std::cout});
```

The correct answer is, “These fragments are ill-formed because the iterator into the input range that \text{ranges::copy} returns would dangle - despite that the program fragment ignores that value - because LEWG asked us to remove the dangling wrapper and make such calls ill-formed.”

In the Ranges TS / revision one of P0896 an algorithm that returns an iterator into a range that was passed as an rvalue argument first wraps that iterator with the \text{dangling} wrapper template. A caller must retrieve the iterator value from the wrapper by calling a member function, opting in to potentially dangerous behavior explicitly. The use of \text{dangling} here makes it impossible for a user to inadvertently use an iterator that dangles.

In practice, the majority of range-v3 users in an extremely rigorous poll of the \#ranges Slack channel (i.e., the author and two people who responded) never extract the value from a \text{dangling} wrapper. We prefer to
always pass lvalue ranges to algorithms when we plan to use the returned iterator, and use `dangling` only as a tool to help us avoid inadvertent use of potentially dangling iterators. Unfortunately, P0896 makes calls that would have used `dangling` in the TS design ill-formed which forces passing ranges as lvalues even when the dangling iterator value is not used.

We propose bringing back `dangling` in a limited capacity as a non-template tag type to be returned by calls that would otherwise return a dangling iterator value. This change makes the program fragments above well-formed, but without introducing the potentially unsafe behavior that LEWG found objectionable in the prior `dangling` design: there’s no stored iterator value to retrieve.

### 6.1 Technical specifications

Modify the `<ranges>` synopsis in [ranges.syn] as follows:

```cpp
namespace std::ranges {
[...]

// [range.range], Range
template<class T>
using iterator_t = decltype(ranges::begin(declval<T&>()));

template<class T>
using sentinel_t = decltype(ranges::end(declval<T&>()));

template<forwarding-range R>
using safe_iterator_t = iterator_t<R>;

template<class T>
concept Range = see below;
[...]

template<Iterator I, Sentinel<I> S = I, subrange_kind K = see below>
requires (K == subrange_kind::sized || !SizedSentinel<S, I>)
class subrange;

// [dangling], dangling iterator handling
struct dangling;

namespace std {
    struct dangling {
        constexpr dangling() noexcept = default;
    }
[...]
}
```

Add a new subclause to [range.utility], following [range.subrange]:

```
23.6.4 Dangling iterator handling

The tag type `dangling` is used together with the template aliases `safe_iterator_t` and `safe_subrange_t` to indicate that an algorithm that typically returns an iterator into or subrange of a `Range` argument does not return an iterator or subrange which could potentially dangle for a particular rvalue `Range` argument which does not model `forwarding-range` ([range.range]).
```

namespace std {
    struct dangling {
        constexpr dangling() noexcept = default;
    }
```

```
template<class... Args>
constexpr dangling(Args&&...)
noexcept {}
};

Example:

vector<int> f();
auto result1 = ranges::find(f(), 42); // #1
static_assert(Same<decltype(result1), dangling>);
auto vec = f();
auto result2 = ranges::find(vec, 42); // #2
static_assert(Same<decltype(result2), vector<int>::iterator>);
auto result3 = ranges::find(subrange{vec}, 42); // #3
static_assert(Same<decltype(result3), vector<int>::iterator>);

The call to ranges::find at #1 returns dangling since f() is an rvalue vector; the vector
could potentially be destroyed before a returned iterator is dereferenced. However, the calls at
#2 and #3 both return iterators since the lvalue vec and specializations of subrange model
forwarding-range. —end example]

Bibliography
