Ranges Design Cleanup
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 combination of N4762 and P0896R3.

1.1 Revision History

1.1.1 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 decltype(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);

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

The following member is declared in addition to those members specified in [move.iterate.elem]:

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

Returns: current.

3 ref-view => ref_view [ref]

The authors of P0896 added the exposition-only view type ref-view (P0896R3 [range.view.ref]) to serve as the return type of view::all ([range.adaptors.all]) when passed an lvalue container. ref-view<T> 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, and users may like to use it as well as a sort of ‘Ranges reference_wrapper’.

3.1 Technical Specifications [ref.words]

— 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]). The expression view::all(E) for some subexpression E is expression-equivalent to:
  - DECAY_COPY(E) if the decayed type of E models View.
  - Otherwise, ref-view[E] ref_view[E] if that expression is well-formed, where ref-view is the exposition-only View specified below.
  - Otherwise, subrange{E} if that expression is well-formed.
  - Otherwise, view::all(E) is ill-formed.
Retitle [ref.view] to “class template ref_view” and modify as follows:

```cpp
namespace std::ranges {
    template<Range R>
    requires std::is_object_v<R>
    class ref_view : public view_interface<ref_view<R>> {
    private:
        R* r_ = nullptr; // exposition only
    public:
        constexpr ref_view() noexcept = default;
        constexpr ref_view(R& r) noexcept;
        constexpr R& base() const;
        constexpr iterator_t<R> begin() const noexcept(noexcept(ranges::begin(*r_)));
        constexpr sentinel_t<R> end() const noexcept(noexcept(ranges::end(*r_)));
        constexpr bool empty() const noexcept(noexcept(ranges::empty(*r_))) requires requires { ranges::empty(*r_); };
        constexpr auto size() const noexcept(noexcept(ranges::size(*r_))) requires SizedRange<R>;
        constexpr auto data() const noexcept(noexcept(ranges::data(*r_))) requires ContiguousRange<R>;
        friend constexpr iterator_t<R> begin(ref_view&& r) noexcept(noexcept(r.begin()));
        friend constexpr sentinel_t<R> end(ref_view&& r) noexcept(noexcept(r.end()));
    };
}
```

Similarly change the class template name in the detailed specification of the operations in [range.view.ref.ops]:

1. `constexpr ref_view(R& r) noexcept;`  
   **Effects:** Initializes r_ with addressof(r).

2. `constexpr R& base() const;`  
   **Effects:** Equivalent to: `return *r_;`

3. `constexpr iterator_t<R> begin() const noexcept(noexcept(ranges::begin(*r_)));`  
   **Effects:** Equivalent to: `return ranges::begin(*r_);` or `return r.begin();`, respectively.

4. `constexpr sentinel_t<R> end() const noexcept(noexcept(ranges::end(*r_)));`  
   **Effects:** Equivalent to: `return ranges::end(*r_);` or `return r.end();`, respectively.

5. `constexpr bool empty() const noexcept(noexcept(ranges::empty(*r_))) requires requires { ranges::empty(*r_); };`  
   **Effects:** Equivalent to: `return ranges::empty(*r_);`
constexpr auto size() const
oxcept(noexcept(ranges::size(*r_)))
requires SizedRange<R>;
    Effects: Equivalent to: return ranges::size(*r_);

constexpr auto data() const
noxcept(noexcept(ranges::data(*r_)))
requires ContiguousRange<R>;
    Effects: Equivalent to: return ranges::data(*r_);

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 std comparison function objects, i.e. class templates that you shouldn’t specialize because they cannot be specialized consistently with the void specializations that you actually should be using?” For the Ranges TS, it was a design goal to minimize differences between 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-template-ing the std::ranges comparison function objects, leaving only concrete classes with the same behavior as the prior void specializations.

4.1 Technical specifications [untemp.words]

In [functional.syn], modify the declarations of the comparison function objects as follows:

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>;
}
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 >=.

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

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

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

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

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

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

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 \( \text{std::forward<T>(t)} == \text{std::forward<U>(u)} \) 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);

\[ \text{std::forward<T>(t)} == \text{std::forward<U>(u)} \]

\[ \text{std::forward<T>(t)} < \text{std::forward<U>(u)} \]

Expects: If the expression \( \text{std::forward<T>(t)} < \text{std::forward<U>(u)} \) 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 \( \text{decltype}((ET)) \) is \( T \) and \( \text{decltype}((EU)) \) is \( U \), exactly one of \( \text{ranges::less} \{ET, EU\} \), \( \text{ranges::less} \{EU, ET\} \), or \( \text{ranges::equal_to} \{ET, EU\} \) shall be true.

Effects:

— If the expression \( \text{std::forward<T>(t)} < \text{std::forward<U>(u)} \) 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.

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

\[ \text{std::forward<T>(t)} < \text{std::forward<U>(u)} \]
using is_transparent = unspecified;
};

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], [algnth.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 [alg.permutation.generators].

5 Reversing a reverse_view [weiv_esrever]

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 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 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 (see cmcstl2/compare/reverse_reverse).

5.1 Technical specifications [sdrow.weiv_esrever]
— Modify the specification of view::reverse in [range.adaptors.reverse] as follows:

1 The name view::reverse denotes a range adaptor object ([range.adaptor.object]). The expression view::reverse(E) for some subexpression E is expression-equivalent to reverse_view(E).

1 If the type of E is a cv-qualified specialization of reverse_view, E.base().

1 Otherwise, reverse_view(E).

6 Exposing exposition-only concepts [expo]

P0896 [specialized.algorithms] provides “rangified” versions of the specialized memory algorithms uninitialized_copy et al. The algorithms are constrained using a family of concepts that refine iterator, sentinel, or range concepts by forbidding some of the required operations to emit exceptions. LWG reviewers were displeased that these concepts are all exposition-only, instead of making them available to users who want to write their own raw memory algorithms.

Aside: There is general uneasiness among LWG reviewers with the amount of exposition-only machinery in P0896. We explained that this is a natural consequence of the addition of Concepts to the language. In C++17 we might have exhaustively repeated the same set of requirements in Requires elements for several library functions, but in C++20 it’s “easy” to define a concept as a handle to that set of requirements and use the concept to directly constrain the several library functions. Obviously not every set of requirements is generally useful and fully-designed to the point that it should be exported to users with a public name, so we end up with exposition-only concepts. LEWG should expect pushback in the future against designs with substantial exposition-only machinery and questions about whether or not consideration has been given to exporting that machinery.

This paper proposes that LEWG reconsider making the concepts in P0896 [special.mem.concepts] exposition only, and provides wording to export those concepts.
6.1 Technical specifications

Modify [memory.syn] as follows:

```cpp
// [specialized.algorithms], specialized algorithms
// [special.mem.concepts], special memory concepts
template<class I>
  concept no-throw-input_iterator NoThrowInputIterator = see below; // exposition only

template<class I>
  concept no-throw-forward_iterator NoThrowForwardIterator = see below; // exposition only

template<class S, class I>
  concept no-throw-sentinel NoThrowSentinel = see below; // exposition only

template<class R>
  concept no-throw-input-range NoThrowInputRange = see below; // exposition only

template<class R>
  concept no-throw-forward-range NoThrowForwardRange = see below; // exposition only

template<class T>
  constexpr T* addressof(T& r) noexcept;

namespace ranges {
  template<
    no-throw-forward-iterator NoThrowForwardIterator I,
    no-throw-sentinel NoThrowSentinel<I> S>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_default_construct(I first, S last);

  template<
    no-throw-forward-range NoThrowForwardRange R>
    requires DefaultConstructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> uninitialized_default_construct(R&& r);

  template<
    no-throw-forward-iterator NoThrowForwardIterator I>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

namespace ranges {
  template<
    no-throw-forward-iterator NoThrowForwardIterator I,
    no-throw-sentinel NoThrowSentinel<I> S>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_value_construct(I first, S last);

  template<
    no-throw-forward-range NoThrowForwardRange R>
    requires DefaultConstructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> uninitialized_value_construct(R&& r);

  template<
    no-throw-forward-iterator NoThrowForwardIterator I>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

namespace ranges {
  template<
    InputIterator I, Sentinel<I> S1,
    no-throw-forward-iterator NoThrowForwardIterator O,
    no-throw-sentinel NoThrowSentinel<O> S2>
    requires Constructible<iter_value_t<O>, iter_reference_t<O>>
    O uninitialized_value_construct_n(O first, O last);
```
uninitialized_copy_result<I, O>
  uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);

template<InputRange IR, NoThrowForwardRange OR>
  requires Constructible<iter_value_t<iterator_t<IR>>, iter_reference_t<iterator_t<IR>>>
  uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
  uninitialized_copy(IR&& ir, OR&& or);

template<class I, class O>
  using uninitialized_copy_n_result = uninitialized_copy_result<I, O>;

template<InputIterator I, NoThrowForwardIterator O, NoThrowSentinel<O> S>
  requires Constructible<iter_value_t<O>, iter_rvalue_reference_t<I>>
  uninitialized_move_result<I, O>
  uninitialized_move(I ifirst, iter_difference_t<I> n, O ofirst, S olast);

namespace ranges {
  template<class I, class O>
    using uninitialized_move_n_result = uninitialized_copy_result<I, O>;

  template<InputIterator I, NoThrowForwardIterator O, NoThrowSentinel<O> S>
    requires Constructible<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_n_result<I, O>
    uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

namespace ranges {
  template<Destructible T>
    void destroy_at(T* location) noexcept;
}
Some algorithms in this subclause are constrained with the following exposition-only concepts:

1. Some algorithms in this subclause are constrained with the following exposition-only concepts:

   template<class I>
   concept no-throw-input-iterator NoThrowInputIterator = // exposition only
   InputIterator<I> &&
   is_lvalue_reference_v<iter_reference_t<I>> &&
   Same<remove_cvref_t<iter_reference_t<I>>, iter_value_t<I>>;

   No exceptions are thrown from increment, copy construction, move construction, copy assignment, move assignment, or indirection through valid iterators.

2. template<class S, class I>
   concept no-throw-sentinel NoThrowSentinel = Sentinel<S, I>; // exposition only

   No exceptions are thrown from comparisons between objects of type I and S.

3. [Note: The distinction between Sentinel and no-throw-sentinel NoThrowSentinel is purely semantic. — end note]

4. template<class R>
   concept no-throw-input-range NoThrowInputRange = // exposition only
   Range<R> &&
   no-throw-input-iterator NoThrowInputIterator<iterator_t<R>> &&
   no-throw-sentinel NoThrowSentinel<sentinel_t<R>, iterator_t<R>>;

   No exceptions are thrown from calls to begin and end on an object of type R.

5. template<class I>
   concept no-throw-forward-iterator NoThrowForwardIterator = // exposition only
   no-throw-input-iterator NoThrowInputIterator<I> &&
   ForwardIterator<I> &&
   no-throw-sentinel NoThrowSentinel<I, I>;

   template<class R>
   concept no-throw-forward-range NoThrowForwardRange = // exposition only
   no-throw-input-range NoThrowInputRange<R> &&
   no-throw-forward-iterator NoThrowForwardIterator<iterator_t<R>>;

7 Use cases left dangling

What does this program fragment do in P0896?

```cpp
std::vector<int> f();
o = 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 `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 `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 `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 `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.

### 7.1 Technical specifications

[Introduce class `dangling` into the `<ranges>` synopsis in `[ranges.syn]`:

```cpp
template<forwarding-range Range R>
using safe_iterator_t = conditional_t<forwarding-range<R>, iterator_t<R>, dangling>;
```

// [range.requirements], range requirements

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

// [range.adaptors.all]
namespace view { inline constexpr unspecified all = unspecified; }```

Add a new subclause to [range.utility], immediately before [range.view_interface]:

#### 23.7.1 class `dangling` [dangling]

The tag type `dangling` is used to indicate that an algorithm that typically returns an iterator into a `Range` argument does not return an iterator into a particular rvalue `Range` argument which could potentially dangle.

[Example:

```cpp
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 vec is an lvalue range and specializations of subrange model forwarding-range, respectively. —end example]

```cpp
namespace std {
    class dangling {
    public:
        constexpr dangling() noexcept = default;
        template<class... Args>
        constexpr dangling(Args&&...) noexcept {
        }
    };
}
```

Bibliography
